Unified Messaging Atop a Cloud of Micro-Objects

Jan Mark Sebastiaan Wams

ISBN 978-90-5335-621-0

The cover was designed by Ridderprint BV (www.ridderprint.nl).
This thesis was printed by Ridderprint BV (www.ridderprint.nl).

VRIJE UNIVERSITEIT

Unified Messaging Atop a Cloud of Micro-Objects

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 18 december 2012 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Jan Mark Sebastiaan Wams

geboren te Rotterdam

promotor: prof.dr.ir. M.R. van Steen

To the loving memory of Noortje.

Contents

Page

1 Massive Messaging **7**
- 1.1 User-to-User Messaging . . . 8
- 1.2 Current Messaging Systems . . . 9
  - 1.2.1 The E-mail Messaging System . . . 9
  - 1.2.2 The USENET Messaging System . . . 14
  - 1.2.3 Instant Messaging Systems . . . 17
  - 1.2.4 The Weblog Messaging System . . . 21
  - 1.2.5 The SMS Messaging System . . . 24
- 1.3 Unified Messaging . . . 28
- 1.4 Messaging System Taxonomy . . . 32
- 1.5 Diffusion of Unified Messaging . . . 35
- 1.6 Social Network Messaging . . . 36
- 1.7 Summary . . . 36

2 A Unified Messaging Model **41**
- 2.1 The Unified Messaging System . . . 42
  - 2.1.1 Missing Services . . . 42
- 2.2 Unified Messaging Model . . . 45
  - 2.2.1 Goals . . . 45
  - 2.2.2 Target and TISM . . . 46
  - 2.2.3 Protection and Identification . . . 46
  - 2.2.4 Taking Control . . . 47
- 2.3 Resource Allocation . . . 47
- 2.4 Mimicking Legacy Messaging System . . . 48
  - 2.4.1 Internet E-mail Imitation . . . 48
  - 2.4.2 USENET Imitation . . . 49
  - 2.4.3 Instant Messaging Imitation . . . 49
  - 2.4.4 Web Logging Imitation . . . 49
- 2.5 New Messaging Paradigms . . . 50
- 2.6 Summary . . . 54

3 Micro-Objects **59**
- 3.1 Distribution Woes . . . 60
- 3.2 A New Approach . . . 61
- 3.3 Micro-Object Internals . . . 62
- 3.4 Example Scenario . . . 64
- 3.5 Design Issues . . . 67
  - 3.5.1 Identifying Data . . . 67

3.5.2 Locating Data . . . 68
3.5.3 Deleting Data . . . 68
3.5.4 Updating Data . . . 69
3.5.5 Protecting Data . . . 70
3.5.6 Replicating Data . . . 71
3.6 Systems Design . . . 73
3.6.1 The cloud server . . . 74
3.6.2 The lib-server . . . 77
3.6.3 "Hello World!" With Micro-Objects . . . 89
3.6.4 Example Extension . . . 91
3.7 The Micro-Object Clusters . . . 94
3.7.1 Managing Clusters . . . 95
3.8 Security and Emergence . . . 105
3.8.1 Security . . . 105
3.8.2 Emergent Behavior . . . 106
3.9 Related Work . . . 107
3.10 Summary . . . 107

4 A Unified Messaging System 113
4.1 Implementation Description . . . 115
4.1.1 Target Design . . . 116
4.1.2 TISM Design . . . 118
4.2 Security . . . 121
4.2.1 Protecting the Target . . . 122
4.2.2 Protecting the TISM . . . 123
4.3 Spam . . . 124
4.3.1 Conventional Measures Against Spam . . . 125
4.4 Unique New Features . . . 129
4.5 Storage versus Transport . . . 129
4.6 Application Programming Interface Design . . . 130
4.7 Future Improvements . . . 130
4.8 Summary . . . 132

5 Micro-Object Experiments 137
5.1 Implemented Replication Policies . . . 138
5.1.1 Base Replication . . . 138
5.1.2 Flooding Replication . . . 139
5.1.3 Interval Replication . . . 139
5.2 Implemented Security Policies . . . 140
5.3 Moping . . . 141
5.3.1 Moping Testing Platform . . . 142
5.3.2 Moping with Base Replication . . . 143
5.3.3 Moping with Flooding Replication . . . 143
5.3.4 Moping with Interval Replication . . . 144
5.3.5 Moping with Mixed Replication . . . 145
5.4 Asynchronous Moping . . . 147
5.4.1 Asynchronous Moping with Flooding Replication . . . 147
5.4.2 Asynchronous Moping with Interval Replication . . . 148
5.4.3 Asynchronous Moping with Mixed Replication . . . 149

5.4.4 Summary 149

6 Conclusions **155**
6.1 Work Related to Unified Messaging 155
6.2 Work Related to Micro-Objects 158
6.2.1 Micro-Objects as Building Blocks 158
6.2.2 A Cloud Of Micro-Objects 163
6.2.3 Future Work 166
6.2.4 Summary 167

A Source Code **169**

Summary

Unified Messaging Atop a Cloud of Micro-Objects

To communicate is human. From the cradle to the grave people try to communicate, by exchanging messages. Tanks to modern technology, there are so many ways to exchange messages, ranging from the trusted old e-mail system to more recent systems like WhatsApp, Facebook chat and Twitter. To research these messaging systems this thesis introduces a taxonomy for messaging systems, followed by an in-depth review of a few well-known systems. Unified messaging is then defined as a messaging system that can mimic the behavior of all messaging systems thinkable within the taxonomy. A two-layered design is proposed to implement unified messaging. The unified messaging layer sits on top of a middleware layer that offers location-agnostic large-scale distributed data-object abstraction or, put more succinctly, sits atop a "data cloud." The middleware layer, called the micro-object layer, is designed as an independent distributed-programming framework. Most distributed-programming frameworks, like CORBA and Java Enterprise Beans, offer a wrapper that transparently transforms application objects into distributed application objects, like a blanket. The micro-object framework, however, offers very small distributed objects from which distributed application objects can be constructed, like LEGO blocks. This thesis further describes the design and implementation of the two layers and offers a performance and performance/resource trade-off analysis and concludes that unified messaging atop a cloud of micro-objects is feasible.

Samenvatting

Universele Bericht Uitwisseling op een Mini-Objecten Wolk.

Communiceren is menselijk. Van de wieg tot het graf proberen mensen te communiceren, ze wisselen berichten uit. Dankzij de moderne technologie zijn er vele manieren om berichten uit te wisselen, van het klassieke e-mail systeem tot de meer recente systemen zoals WhatsApp, Facebook chat en Twitter. Om onderzoek te kunnen plegen op bericht uitwisselssystemen word in deze dissertatie een taxonomie geïntroduceerd gevolgd door een grondige beschouwing van een aantal bekende systemen. Universele bericht uitwisseling word vervolgens gedefinieerd als een bericht uitwisselssysteem dat alle functionaliteit heeft van alle bericht uitwisselsystemen bij elkaar die in de taxonomie passen. Een gelaagd implementatie ontwerp voor universele berichtuitwissling word gepresenteerd. De universele bericht uitwissellaag zit boven op een tussenlaag die zorgt voor lokatie agnostische, groot grootschalige, gespreide gegevens objecten abstractie, of wat korter, op een "gegevens wolk". Deze tussenlaag, mini-objecten-laag genaamd, is apart ontwikkeld als een onafhankelijke werkruimte voor gespreid programmeren. De meeste van dit soort werkruimten, zoals CORBA en Java Enterprise Beans, bieden een wikkel aan die op een transparante wijze applicatieobjecten om vormt tot gespreide applicatieobjecten, als een deken. De mini-objecten-werkruimte daar in tegen, biedt hele kleine gespreide objecten aan waarmee gespreide applicatieobjecten kunnen worden geconstrueerd, net als LEGO blokjes. Deze dissertatie beschrijft vervolgens het ontwerp en de implementatie van de twee lagen en biedt een prestatie en prijs/prestatie-verhoudingsanalyse en concludeert dat universele berichtuitwisseling op een wolk van mini objecten alleszins haalbaar is.

❧ *"What is big and hardly researched?"* This intriguing question was put forward at my first meeting with my promoter. I sat silently and pondered the question. A really big thing usually is a blissful ignorant's fact of life. Like supermarket logistics, clean tap water, or font kerning. However, specialists do research those topics. Something has to be hiding in über ubiquity to hardly be researched. *"Like E-mail,"* I remember saying out loud. My promoter instantly iterated several E-mail related research topics. All were about accomplishing something, none about what E-mail really was. *"But, what constitutes E-mail?"* my return question was. The discussion spun into a debate on electronic messaging systems and how many kinds there were. Like counting childhood songs, it was unstructured and surprisingly ever-elongating. A structured discussion needed a classification of electronic messaging systems. None could be found quickly. So my first assignment was to find or create a taxonomy for electronic messaging systems. It seemed trivial until I actually tried. Also, it made me wonder about the large number of electronic messaging systems. It led me to a moderate ambitious research question: *"Can a single messaging system cover the whole taxonomy?"*

Chapter 1

Massive Messaging

> "When the mail was being developed, nobody thought at the beginning it was going to be the smash hit that it was."
>
> *Frank Heart*, the director of the 1969 ARPANET building team.

To communicate is human. From the cradle to the grave people try to communicate, by exchanging messages. This dissertation is about the exchange--not the content--of messages. Therefore the term "messaging" rather than "communicating" is used. In line with this, a *messaging system* is defined as a system that transports a digitized message from a user-input device over a network to one or more user-output devices. By this definition, the cell-phone SMS (Short Message Service)[1] system is a messaging system, and POTS (Plain Old Telephone Service) is not, even though VoIP (Voice over Internet Protocol) telephone systems are. There are many messaging systems and one could wonder, since they are all used to exchange messages, if they are all more or less equivalent. These messaging systems might only differ in infrastructure or payment method. These practical differences clearly exist, however, in this chapter I will demonstrate that:

> *Messaging systems can be fundamentally different and they can be systematically analyzed and classified.*

Only messaging systems capable of handling massive numbers of users sending massive numbers of messages, have been considered for this writing. These mes-

[1] On page 184 a list of acronyms and their meaning is supplied.

System	Type	Protocol
E-mail	telegram	SMTP
USENET	bulletin board	NNTP
Weblog	public diary	HTTP
Twitter	micro blog	Kestrel
BitTorrent	telefax	BitTorrent
AIM	chatroom	AIM
SMS	paging	CIMD
I-mail	telegram	i-mode

Table 1.1: A few examples of messaging systems. The "type" column contains an attempt to classification and the column headed "protocol" contains one of associated protocols.

saging systems typically handle hundreds of millions of users exchanging a total of billions of messages per day. Messaging systems of this size are sometimes referred to as large-scale messaging systems or *massive-messaging systems*.

1.1 User-to-User Messaging

There are many user-to-user messaging systems. One of the biggest is Internet E-mail (Electronic Mail). However, E-mail is not the only messaging system on the Internet and the Internet is not the only infrastructure to feature messaging systems. Lately, the growth in popularity of cellular handheld devices has triggered the development of many messaging systems like WhatsApp (WhatsApp Messenger), SMS, MMS (Multimedia Message Service), I-mail (Information mail), and messaging systems based on WAP (Wireless Application Protocol).

In Table 1.1 a few well known messaging systems are listed with their respective type and supporting protocol. Classification of type and protocol in this table is informal. For example, the table suggests a weblog is a public diary, this does not do justice to the feedback options that most weblog systems have. As another example, SMS uses CIMD (Computer Interface to Message Distribution) as main protocol, but SMPP (Short Message Peer-to-Peer Protocol) or SS7 (Signaling System 7) could equally well be listed. Without a taxonomy it is even unclear if a peer-to-peer file sharing system like BitTorrent actually

classifies as a messaging system.

In short, what is needed is a more precise description of what a messaging system is and how it can be classified. Despite considerable debate, there is no consensus how to compare or categorize messaging systems. It is not obvious how to compare the fax system to USENET (UNIX USEr NETwork). There is not even a common set of topics or keywords for papers on messaging systems. To this end, a taxonomy for messaging systems will be presented in Section 1.4. First, laying the ground work, an informal reconnaissance of the field of messaging systems will be presented.

1.2 Current Messaging Systems

Depending in the definition, the most dominant messaging system is probably Internet E-mail. Measured in bandwidth, the BitTorrent file sharing system is larger than the E-mail system. However, the total number of E-mail messages sent per day vastly outnumbers the number of files exchanged through BitTorrent. Also the number of E-mail users is much higher than the number of BitTorrent users.

Not only is E-mail big, it is also archetypal in the sense that it resembles the postal system that finds its roots in organized courier service like the one the Pharaohs used as early as 2400 BCE or King Ahasuerus (Xerxes) according to the Old Testament[2].

This section, therefore, starts off with a description of the E-mail messaging system. The E-mail system is then used as a reference point in the ensuing description of USENET, IM (Instant Messaging), weblog, and SMS messaging systems. These four have properties not found in E-mail, which help explain why they exist as separate messaging systems.

1.2.1 The E-mail Messaging System

When, in 1971, Ray Tomlinson sent the first E-mail message across one of the precursors of the Internet called ARPANET (Advanced Research Project Agency Network), few foresaw that his logical extension to intra computer E-mail would become the biggest homogeneous messaging system in the world, even though Tomlinson's program `SNDMSG` quickly became responsible for more than half of the traffic on ARPANET. It might be argued that E-mail was Internet's first *killer application* (that is, the principal reason for a user to get on

[2]King James Bible--Esther 8:10 And he wrote in the King Ahasuerus' name, and sealed it with the king's ring, and sent letters by posts on horseback, and riders on mules, camels, and young dromedaries…

the Internet). Internet E-mail is starting to rival POTS as the preferred means of interpersonal contact.

For 2012, the Radicati Group estimated the number of E-mail messages sent to be around 144.8 billion *per day*, with about 90% of those being unsolicited messages. They also estimated the population of active E-mail accounts at 3.4 billion for 2012 and projected a growth rate of over 6% for the next four years, probably one in four of the world's human population actively uses E-mail. This would seem to indicate that every E-mail user would be sending out dozens of messages a day. Due to spam (not the luncheon meat canned by Hormel Foods) the mean number of messages will be far lower than that. Regardless, the E-mail system is big, ubiquitous and still growing. In the USA over 90% of the Internet population uses E-mail according to the 2009 *Internet Investment Guide* of J.P.Morgan. Although rivaled by applications such as IM, peer-to-peer file sharing systems, and cellular messaging systems, Internet E-mail is still, the second most used messaging system the world over, after the SMS system. A notable exception is China, according to J.P.Morgan, where "only" 57% of the Internet population use E-mail but 81% use IM (versus 39% in the USA).

The basic model for E-mail is simple: a user sends a message to one or more explicitly addressed recipients, where it is subsequently stored in the recipient's mailbox for further processing. One of the main advantages of this model, to the user, is the asynchronous nature of communication: the recipient need not be online when a message is delivered to their mailbox, but instead, can read the message at any convenient later time.

Principal Operation

The basic organization of the E-mail system is shown in Figure 1.1(a) and consists of several components. From a user's perspective, a *mailbox* is conceptually the central component. A mailbox is simply a storage area that is used to hold messages that have been sent to a specific user. Each user generally has one or more mailboxes from which messages can be read and removed. The mailbox is accessed by means of an MUA (Mail User Agent), which is a management program that allows a user to, for example, edit, send, and receive messages. Common ones include Eudora, Outlook Express, Thunderbird, Entourage, Apple Mail, /bin/mail, Elm, Mutt, and Pine.

Once a message has been composed with the MUA, it has to be sent out. To this end, the MUA generally contacts a local MSA (Message Submit Agent) that temporarily queues outgoing messages. The actual intermachine exchange of E-mail messages is taken care of by a mail server, called MTA (Message Transfer Agent). The MTA at the sender's site is responsible for removing messages that

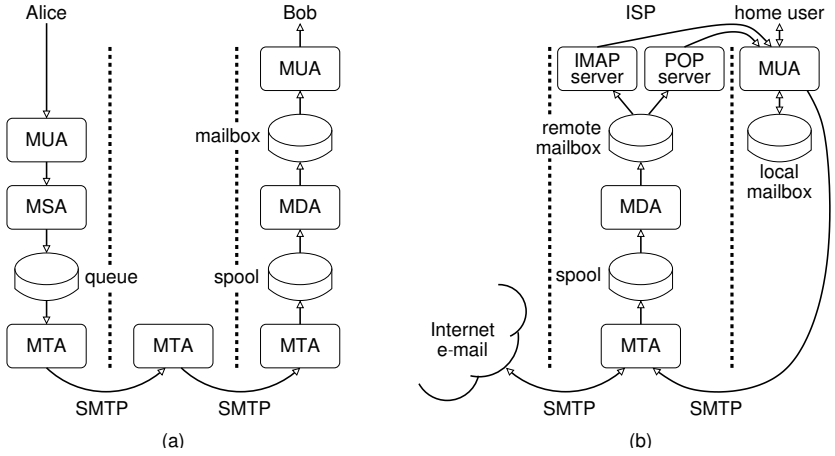

Figure 1.1: (a) The general organization of E-mail. (b) How E-mail is supported by an ISP.

have been queued by the MSA, and transferring them to their destinations, possibly routing them across several other MTAs. At the receiving side, the MTA spools incoming messages, making them available to the MDA (Message Delivery Agent). The latter is responsible for moving spooled messages into the mailboxes of the proper users.

Assume that Alice at site A has sent a message m to Bob at site B. Initially, this message will be stored by the MSA at site A. When the message is eventually to be transferred, the MTA at site A will set up a connection to the MTA at site B and pass it message m. Upon its receipt, this MTA will store the message for the MDA at B which, in turn, looks up the mailbox for Bob to subsequently store m. MTAs exchange messages according to SMTP (Simple Mail Transfer Protocol) as specified in [41].

Note that this organization has a number of desirable properties. In the first place, if the mail server at the destination's site is currently unreachable, the MTA at the sender's site will simply keep the message queued as long as necessary. Only in extreme cases (e.g., an MTA is offline for several days), will an error E-mail message be sent back to the listed sender. As a consequence, the actual burden of delivering a message in the presence of unreachable or unavailable mail servers is mostly hidden from the E-mail users.

Remote Access

The organization as sketched in Figure 1.1 assumes that the user agent has continuous (local) access to the mailbox. In many cases, this assumption does not hold. For example, many users have an E-mail account at an ISP (Internet Service Provider). However, it is not customary for a user to physically visit an ISP's premises to read or send E-mail. Rather, mail sent to a user is initially stored in the mailbox located at the ISP, and made remotely accessible by a special server, as shown in Figure 1.1(b).

The remote access server essentially operates as a proxy for the MUA, allowing the user access from many different places. There are two models for its operation. In the first model, which has been adopted in (version 3 of) the POP (Post Office Protocol) standard specified in [28], the remote access server transfers a newly arrived message to the user, who is then responsible for storing it locally. Although POP allows to keep a transferred message stored at the ISP, it is customary to configure user agents to instruct the server to delete any message that the agent had just fetched. This setup is often necessary due to the limited storage space that an ISP provides to each mailbox. However, even when storage space is not a problem, POP provides only minimal mailbox search facilities, making the model not very popular for managing messages. It used to be dominant.

As an alternative, there is also a model in which the access server does not normally delete messages after they have been transferred to the user. Instead, it is the ISP that takes responsibility for mailbox management. This model is supported by (version 4 of) IMAP (Internet Message Access Protocol), as is specified in [23]. In this case, the access server provides an interface that allows a user to browse, read, search, and maintain his mailbox from different devices at different locations. IMAP is particularly convenient for mobile users, because IMAP can be supported by cellular devices.

Another example of the latter model is called *webmail*. Webmail allows users access to their remote mailbox, using a WWW (World Wide Web) browser. Usually a webmail server uses IMAP to access E-mail messages, and generates HTML (HyperText Markup Language) web pages displaying them. Note that with the pervasiveness of web browsers, webmail comes close to providing roaming access to E-mail. Notable webmail providers are Microsoft (@hotmai.com, @outlook.com, @live.com), Mail.ru (@mail.ru), Google (@gmail.com), Ceno Technologies (@ceno.cn), Apple (@mac.com, @me.com, @icloud.com), Yahoo! (@yahoo.com) and AOL (@aol.com, @aol.de).

```
; <<>> DiG 9.1.0 <<>> mx cs.vu.nl
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43753
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 4, ADDITIONAL: 3

;; QUESTION SECTION:
;cs.vu.nl.                      IN      MX

;; ANSWER SECTION:
cs.vu.nl.               86069   IN      MX      1 tornado.cs.vu.nl.
cs.vu.nl.               86069   IN      MX      2 zephyr.cs.vu.nl.

;; ADDITIONAL SECTION:
tornado.cs.vu.nl.       86069   IN      A       192.31.231.152
zephyr.cs.vu.nl.        86069   IN      A       192.31.231.66
```

Figure 1.2: Response to a DNS query using `dig MX cs.vu.nl` (edited).

Naming

To enable the transfer of messages a scheme for addressing the source and destination is necessary. For Internet E-mail, an address consists of two parts: the name of the site to where a message needs to be sent which, in turn, is prefixed by the name of the user for which it is intended. These two parts are separated by an at-sign ("@"). Given a name, the E-mail system should be able to set up a connection between the sending and receiving MTA to subsequently transfer a message, after which it can be stored in the addressed user's mailbox. In other words, what is required is that an E-mail name can be *resolved* to the network address of the destination mail server.

Resolving an E-mail name requires support from the Internet DNS (Domain Name System) [5, 47]. Consider sending an E-mail to an address `JohnD@cs.vu.nl`. In this example, `JohnD` identifies the user at site `cs.vu.nl`. To send a message, it is necessary to identify a mail server that can handle incoming E-mail traffic. For Internet E-mail, DNS allows to store such information in what are known as MX (Mail Exchange) records.

For example, using a program called dig (Domain Information Groper), a DNS query requesting an MX record for `cs.vu.nl` returns the answer shown in Figure 1.2.

The most important part of the response is the answer section (shown in boldface in Figure 1.2), which states that there are two mail servers for `cs.vu.nl`. The preferred mail server is named `tornado.cs.vu.nl`, while a secondary server named `zephyr.cs.vu.nl` is also available. To initiate an SMTP session, the sender's MTA usually sets up a TCP (Transmission Control Protocol) connection to the preferred MTA at the destination site, for which it needs the

server's IP address. In the example, this would require resolving the name `tornado.cs.vu.nl`, which is, in principle, done by means of another DNS query. DNS anticipates such additional queries when asked for an MX record and includes an *additional section* containing the IP (Internet Protocol) addresses of the returned mail servers thus avoiding another query.

Once the message has been transferred to the destination MTA, it is the task of the latter to resolve the user name that is part of the E-mail address to the appropriate mailbox. How this user-name resolution is done is not prescribed by SMTP.

Summary of Properties

The E-mail messaging system offers delivery of a message into the receiver-specific storage of any number of pre-known receivers. Once received, messages do not expire. This cryptic description will be restated more formally after the introduction of the messaging taxonomy in Section 1.4.

1.2.2 The USENET Messaging System

USENET, also called *NetNews*, gained its popularity as part of UUCPNET (UNIX-to-UNIX Copy Protocol NETwork), a logical network mainly consisting of many computers that used POTS and modems for message exchange. The USENET model is that of an electronic bulletin board: messages are put up on the board to be read and reacted to by others. In USENET, messages are referred to as *articles* that are *posted* in a specific *newsgroup*. A newsgroup is thus a collection of logically related articles and forms the electronic representation of a bulletin board. A nontechnical overview of network news is given by Comer [22].

A user provides the USENET system with the name of a newsgroup in order to read articles. The *header* of any new article that has not yet been read by the user is then transferred to the user. If the user wants to read the entire article, he will request for the transfer of the article's *body*. After reading an article, a user can respond by posting a reaction in that same newsgroup (which again appears as just another article). Cross postings by which an article refers to an article in a different newsgroup is also possible.

Note that users do not actively delete articles. However, to prevent postings from consuming storage indefinitely, system administrators generally remove articles after some time. Unlike E-mail where messages are permanently stored until explicitly deleted by a recipient, this policy makes news articles impermanent to the recipient.

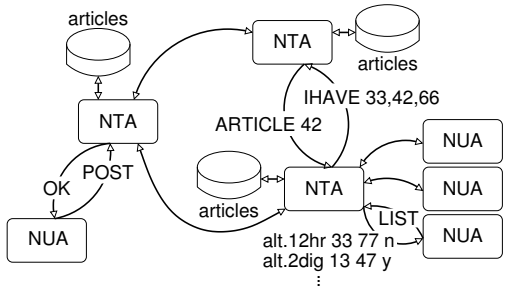

Figure 1.3: The general organization of network news.

Principal Operation

The core of the network news system is formed by a huge collection of news servers that are spread across thousands of different sites. A news server, also referred to as an NTA (News Transfer Agent), is capable of receiving, sending, and storing articles. The basic organization is shown in Figure 1.3. A client, called an NUA (News User Agent), connects to a news server to read and post articles for one or several newsgroups. Likewise, servers connect to each other to exchange articles as will be discussed in more detail below. Although the figure suggests a principal difference between clients and servers, no such difference actually exists. In fact, the protocol that is used between a client and server and the one used between two servers is the same. All information exchange follows NNTP (Network News Transfer Protocol), as specified in [37]. A news server must connect to one or more existing *news feeds*, which are just other NTAs that are willing to exchange articles. In many cases, a news feed is operated by a separate organization such as an ISP. When a news server contacts a news feed, it requests the transfer of new articles. There are several operations available to establish such a transfer of which some important ones are listed in Table 1.2. The transfer protocol is relatively simple and has not been changed since its specification in 1986. However, practice has shown that extensions and deviations from the original specification were needed. In particular, the communication between servers, and that between a client and a server are different enough to warrant further refinements, effectively leading to two very similar, yet different protocols. These refinements are described in [11].

Naming

An important difference between USENET and E-mail is that there is no need to explicitly name and lookup news servers. Instead, the address of a news feed

Operation	Description
LIST	Returns a list of newsgroups available at the callee with each entry identifying the first and last article in that group.
GROUP	Makes a specified group "current," and returns an estimate of the number of articles at the callee in that group.
ARTICLE	Transfers (to the caller) a specified article in the current group.
POST	Tells the callee that an article has been posted at the caller.
IHAVE	Tells the callee that a specific article is available to be sent.
NEWNEWS	Returns a list of articles that have been posted at the callee in specific news groups.
NEWGROUPS	Returns a list of newsgroups that have been created at the callee.

Table 1.2: Commonly used operations to establish the transfer of articles between two news programs.

is assumed to be known at the time a news client or server is configured so that its address can be readily used to setup an NNTP session. Jointly, these sessions ensure that articles are *flooded* through the network consisting of USENET servers. In contrast, for E-mail it is necessary to devise a naming scheme by which users and mail servers can be looked up at runtime. This naming scheme is needed to support the *point-to-point* communication in E-mail systems.

Naming in news therefore restricts itself to newsgroups, and implicitly also articles. In particular, it is important to have a suitable naming scheme for the tens of thousands of newsgroups that currently exist. To this end, a hierarchical naming scheme has been devised that is simple, yet flexible enough to support a large number of newsgroups. A newsgroup name is a series of strings separated by a dot, such as `comp.os.research`. In this example, `comp` identifies the broad category of over one thousand newsgroups related to computer science, which is further divided into over one hundred and fifty newsgroups dealing with operating systems (`os`) and, in particular, the one containing articles on research in this area (`research`).

Each article has a unique identifier consisting of two parts separated by the at-sign. An example of such an identifier is `3e1ed38c$1@news.cs.vu.nl` (see also [31]). The second part identifies the host where the article was first entered into the news system, in this example `news.cs.vu.nl`. The first part is a unique

identifier normally generated by the host named in the second part (and hidden from the user). In principle, an article's identifier is globally unique and is never reused: it is a so-called, true identifier [70].

Summary of Properties

The USENET messaging system offers delivery of a message that does expire, to a number of class-specific storage spaces, to any user interested in its class. These properties differ greatly from the properties of E-mail, as listed at the end of Section 1.2.1.

1.2.3 Instant Messaging Systems

One of the upcoming means of user-centric communication across the Internet is Instant Messaging. The Radicati Group estimates that in 2012 about 2.7 billion IM accounts will be active with and estimated yearly growth rate of 6% for the next four years. The model underlying instant messaging is that of *synchronous communication*: a message can be successfully transferred only if the destination is willing to receive it at the time it is sent. In many other respects, instant messaging strongly resembles E-mail and their user interfaces are sometimes integrated into a single messaging (web) client. One of the first one-to-one instant messaging system was called "term-talk" and ran on the Plato system as early as 1973 [71]. One of the earliest full-blown instant messaging systems appeared in MIT's Athena system [12]. On the Internet IM became popular due to a program called IRC (Internet Relay Chat), described in [48] and updated in [35] and [36]), but became really popular with the introduction of ICQ (I seek you). Currently, there are many instant messaging clients, some are integrated into other services like social network websites or VoIP applications. Instant messaging services are mainly provided by large organizations such as Skype, Facebook, Google, and Microsoft. An instant messaging system is generally paired with a component, called a *presence-information service*, which is used to inform users of the presence of a user-specific list (often referred to as "buddy list") of other users (see also [25]). Such a service allows a user to see whether users on his list are on-line. When a user logs in, his presence is published and forwarded to subscribers of that information. Likewise, a user can indicate that he is temporarily not reachable, or has logged off. Managing presence information is increasingly becoming an important issue as it strongly affects the privacy of publishers. Below, I will return to presence information in more detail.

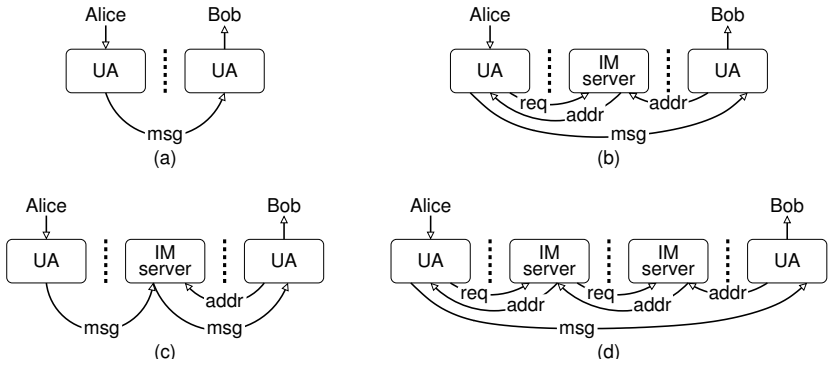

Figure 1.4: Setting up an instant messaging connection: (a) directly, (b) through a central server, (c) centralized, including messaging, and (d) through different servers.

Principal Operation

The principal operation of an instant messaging service is quite simple. In all cases, a *channel* between the communicating parties has to be set up first. Consider the case where Alice wants to communicate with Bob. If Alice has Bob's IP address then, in principle, she can set up a channel directly to Bob's UA (User Agent) as shown in Figure 1.4(a).

The main drawback of this approach is that Bob's contact address must be fixed (i.e., the address where Alice can reach Bob), but also that Alice can set up unsolicited channels to Bob. To alleviate these problems, many instant messaging services adopt the scheme shown in Figure 1.4(b). In this case, a central server keeps track of online clients (whose contact address may be different each time they come online). Alice sends a setup request to the server which subsequently returns Bob's address, possibly after checking whether Alice is authorized to set up a channel to Bob. Note that the central server may also be used as an intermediary for *all* communication between Alice and Bob, that is, including the instant messages sent between them, as shown in Figure 1.4(c).

The obvious drawback of the central server is that it forms a potential bottleneck. This centralized approach, even when multiple servers are used, is recognized as one of the main scalability problems in IRC. To circumvent these scalability problems, several instant messaging servers can be used, as shown in Figure 1.4(d). In this solution, Alice contacts a local instant messaging server and requests a communication channel to Bob. Her local server then queries other servers to find the server that controls connections to Bob (or new servers to query), and requests a communication end point to Bob's client. After the

proper security checks have been made and Alice is indeed found to be authorized to contact Bob, Bob's address is returned to allow the setup of a connection.

This use of a distributed instant messaging service scales well as only the servers that are local to Alice and Bob need to assist in setting up a connection. However, it also introduces a lookup problem, because the server local to Alice needs to locate Bob's local instant messaging server. A simple solution, but one that has not been widely deployed yet, is to follow the same approach as in E-mail. In principle, every site makes use of a single instant messaging server and users simply identify themselves by their E-mail address. Assume Bob's E-mail address is `bob@cs.vu.nl`. When Alice wants to contact Bob, her (well known) local instant messaging server queries the DNS for the instant messaging server at `cs.vu.nl` using a SRV (SeRvice) record request, much like the E-mail system uses the MX record. The XMPP (eXtensible Messaging and Presence Protocol) is a real life example of a protocol that uses the SRV record this way. The addresses used by XMPP resemble E-mail addresses with an optional additional resource identifier. For example one of Alice's XMPP addresses could look like: `alice@cs.vu.nl\slashiphone`.

So far, it has been silently assumed that instant messaging takes place only between pairs of individuals. In general, this need not be the case. Two different forms of multiparty instant messaging exist. First, setting up a connection between two parties can easily be extended by inviting another party, leading to an ad hoc group, or *chat session*. In this case, each message is sent to all members participating in the session. An invited party can *join* the session, and any joined party can later *leave* again. A session dissolves when the last member leaves.

The second type of multiparty instant messaging is through so-called, *chat rooms*, which are effectively permanent sessions. To enter a chat room, a user needs to set up a connection to a well known server that handles all communication for that chat room, effectively leading to the communication scheme shown in Figure 1.4(c). Each message sent to the server is multicast to every other client that has entered the chat room. Unlike ad hoc groups, chat rooms continue to exist even after the last member has left. By their nature, a chat room is useful for online discussions on a very specific subject, and this is indeed the way that they are generally organized.

Presence-Information Service

As mentioned above, an important component of an instant messaging service is a service that provides presence information. In a minimalistic approach, such a

Status	Alert	Description
OFFLINE	No	No IM client is currently running at the recipient.
ONLINE	Yes	The recipient's IM client is currently running.
AWAY	Yes	The client is running, but cannot accept invitations.
BUSY	No	The client is running, but ignores invitations.

Table 1.3: Examples of different states maintained by a presence-information service. The column *alert* indicates whether the recipient is notified when a setup request arrives.

service merely reports whether a user is online or offline, allowing an initiator to see whether it makes sense to even try to set up an instant messaging connection. However, presence information can, and often is, extended with other possible states, as shown in Table 1.3.

Many variations on these states exist. For example, some presence services automatically switch a recipient from *online* to *away* when there has been no interaction with the instant messaging client for some while. Likewise, when an invitation is sent out to a recipient who is currently *busy*, the inviting client may receive a message telling that the other other party does not want to be disturbed.

It is not difficult to see that where instant messaging by itself is relatively simple, a presence-information service can easily grow into a sophisticated and complex part of an instant messaging service. Following the general architecture as described in [25], a presence-information service may also provide the means to send *notifications* when a client's status changes. Such a notification may be useful, for example, when Alice wants to contact Bob as soon as he can accept invitations again.

Despite its attractiveness, the real problem with this functionality starts when thinking about security. In effect, Alice *subscribes* to notifications concerning state changes of Bob. Although it may seem obvious that Bob should be in full control of permissible subscriptions, practice shows that this is not always the case. However, as also laid down in [24], a client should always be in full control concerning who is allowed to send instant messages, and who is allowed to subscribe to presence-state changes. This model has been adopted by the IETF working group on XMPP.

In essence, before Alice can subscribe to presence information concerning Bob, XMPP requires that she sends Bob a request for subscription. If this re-

quest is granted, Bob can pass her the appropriate credentials by which she can obtain a subscription at the presence-information service. Bob, in turn, can always request the presence service to unsubscribe Alice.

How simple this model may seem, it has severe implications for the design and implementation of a presence-information service. The simplest situation is when the presence service is implemented as a single (trusted) centralized server. In that case, managing subscriptions boils down to checking lists of subscribers and sending notifications as needed. However, dealing with a distributed presence-information service, essentially, is equivalent to dealing with a general publish/subscribe system. Having to manage many users who may be geographically widely dispersed, scalability problems suddenly become paramount and obvious solutions do not exist (see for example [17]). However, if a system-wide unique token is shared by all interested parties, a publish/subscribe peer-to-peer system like SCRIBE can be used (see [18]).

Naming

Naming is generally straightforward in instant messaging systems and mainly concerns identifying users. For this reason, systems are gradually adopting the E-mail naming scheme. In the case of chat rooms, instant messaging service providers generally offer a list of topics for which a chat room is hosted. By selecting a topic, a user is then allowed to join a chat session. Naming in such cases is therefore implicit and of less importance than with core instant messaging.

However, naming in many popular instant messaging systems is still much of a nuisance. In particular, several systems such as ICQ simply provide a unique (long) number that is to be used as ID. The drawback of using these numbers is similar to using network IP addresses instead of host names: they are difficult for humans to remember. To circumvent problems, users simply build local lists of aliases for those people they contact regularly.

Summary of Properties

A typical IM system offers delivery of a transient message, to a class-specific storage space, shared by two or more users. These properties differ greatly from the properties of E-mail and somewhat from the properties of USENET.

1.2.4 The Weblog Messaging System

One of the fastest growing messaging forms today is web logging, or simply *blogging* [14]. A weblog, or *blog*, can be viewed as a unidirectional form of one-

to-many messaging: a user maintains a log of messages that others can generally only read, much like an online diary. However, many weblogs allow for readers to react to messages. Web logging can be considered analogous to columns and commentaries in newspapers. Also, many organizations have in-house weblogs. It is hard to pinpoint exactly when the first homepage turned into a weblog but from 1998 till 2002, blogging was largely unknown. From 2002 till 2005, the number of *bloggers*, as they are normally called, grew from thousands to millions, and from 2005 onwards, the total number of weblogs probably has to be measured in tens to hundreds of millions. Precise numbers are difficult to obtain, however some nonscientific sources [51] count (end 2011) 39 million Tumblr blogs and 70 million WordPress blogs alone and automated blog searcher BlogPulse.com claims to have counted a over 147 million blogs up late 2010. Experts seem to agree that blogging is growing more and more popular, probably due to the existence of tools and sites that allow small personal weblogs to be set up almost effortlessly. Most weblogs have a small reader group. However, a few high-volume weblogs serve a huge number of readers. For example, over the course of 2002, the weblog Drudge (`drudgereport.com`) served around a billion pages. The weblog Slashdot (`slashdot.org`) also serves millions of pages per day with "news for nerds." Needless to say that high-volume weblogs usually have multiple editors, and that sophisticated distributed-message moderating takes place to keep the volume usable. With the increasing popularity of RSS (Really Simple Syndication), allowing weblogs to syndicate their content and users to access weblog information in an uniform way, the *blogosphere*, as the total of weblogs is also referred to, is becoming more and more homogeneous.

Nowadays creating a dedicated weblog can be simplified by using a so-called, blog CMS (Content Management System). To start blogging, one only has to set up a DNS domain and a blog CMS, as shown Figure 1.5(a). A blog CMS usually consists of a front end, and a back end. The backend is used by the author(s) to add new entries and manage the system. The front end is what the ordinary user sees, and uses to post comments. An example of such a weblog on a dedicated domain is `sinteur.com`. It utilizes the WordPress CMS. Using a weblog resembles using the worldwide web, as shown in Figure 1.5 (a). With the advent of *blog service providers*, it is even easier to start a weblog. Now a user does not even have to set up a CMS and a dedicated DNS domain, as shown in Figure 1.5(b). One of the bigger sites that allows everybody with a web browser to easily maintain a weblog, is Google-owned Blogger (`blogger.com`). According to Blogger, over a million people have used their service to start a weblog, and subscriptions show an exponential growth. An example of a Blogger hosted weblog is `aap.blogspot.com`.

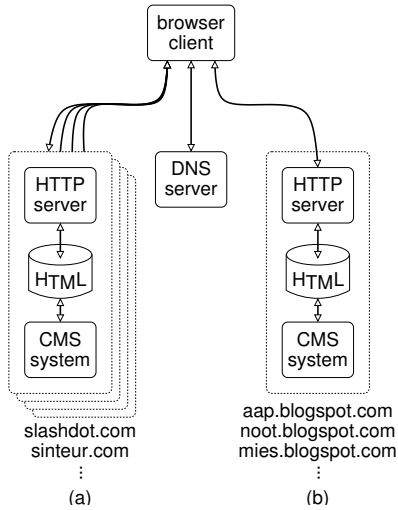

Figure 1.5: Weblog Messaging System, (a) example dedicated domains, (b) example service provider.

Principal Operation

The principal operation of web logging is extremely simple: a blogger simply publishes material on a single site that can be read by anyone accessing that site. Many tools are available that ease the process of updating and managing published material, effectively hiding the technical intricacies related to web servers. An important difference with all messaging systems discussed so far, is that there are, in principle, no immediate recipients. All material related to a blog is conceptually permanently published at a single website that needs to be visit regularly to keep track of changes (i.e., the arrival of new articles) or an RSS feed could be used to pull the changes automatically at regular intervals. As of version 2.0 the RSS standard also specifies a push mechanism that fundamentally changes the blog system to a mailing-list like system.

Summary of Properties

A typical weblog-messaging system offers delivery of a message, that does not expire, to a sender-specific storage space, to any WWW user. These properties differ some what from the properties of E-mail and the other messaging systems discussed above.

1.2.5 The SMS Messaging System

Worldwide, a total of 4.5 trillion SMS messages were sent in 2009 or just under 2.7 messages a day for all 4.6 billion active mobile subscriptions. [3] Number wise, the SMS system is the biggest messaging system in the world, by far. By comparison, as stated above, in 2009 about 1.4 billion E-mail users sended out 0.25 trillion messages and in 2008, about 0.6 billion IM users send out less than 0.005 trillion messages. Money wise, the SMS system probably is also the biggest, worldwide. It is hard to measure since SMS messages are usually paid piecemeal or in bundles, where most E-mail and IM messages are paid for indirectly through an Internet connection fee. Worldwide the average consumer price for an SMS message will be around 10 cent (USD), while the cost of delivery approaches zero. Telecom operators usually charge half of that for delivery of SMS messages that originate outside their network.

Cellular messaging and Internet messaging are becoming increasingly intertwined and there is little doubt that they will integrate further in the future.

One thing cellular messaging solves differently from POTS and Internet messaging, is reaching a roaming user. With cellular networks, users take the messaging device with them; with other messaging systems, the user roams from messaging device to messaging device. This is exactly as with voice communication. A cellular device is carried while roaming and with POTS voice communication, the user roams from phone to phone. With the rising popularity of cell phones, the popularity of pay phones is going down. No doubt a similar trend will be seen with messaging. The downside of having a user device roam with the user, is the single point of failure it introduces. With the loss of a cellular device comes a substantial diminishing of communication ability. On the other hand, when a pay phone is out of order, the loss of communication ability is distributed amongst many users. In Section 1.3 a general unifying model for messaging is introduced that does not have this trade-off.

Principal Operation

The SMS system is a *store-and-forward*, *end-point delivery* messaging system, predominantly implemented on top of MAP (Mobile Application Part). There are two similar standards, the American IS-41 MAP (or ANSI-41 MAP) and the international GSM MAP. In turn, MAP sits on top of TCAP (Transaction Capabilities Application Part) on top of the SS7 network layer, as shown in Figure 1.6. In the past telephony used SS5 (Signaling System 5) *in-band signaling* for call setup and teardown. Information was sent by playing special tones into the telephone lines (also called *bearer channels*). For security and other reasons,

SS7 based SMS	
SS7 layer	**OSI layer**
(GSM / IS-41) MAP TCAP	7. Application
SCCP / MTP Level 3	3. Network
MTP Level 2	2. Data link
MTP Level 1	1. Physical

Figure 1.6: SS7 protocols stack for SMS message delivery. The approximate equivalent ISO/OSI model layers are listed on the right.

the SS5 in-band system gave way to SS7. With SS7, data exchange is done *out-of-band*, on a separate data channel. The ITU (International Telecommunication Union) defined SS7 as a packet-switching four-layer stack, resembling the seven-layer ISO/OSI network stack. For an overview of SS7 see [26]. Note that using packet switching was a radical break with the traditional CSD (Circuit Switched Data) technology applied by older telephone systems. Soon the SS7 out-of-band data channel, was used for more than just call setup and teardown. This data channel proved a perfect infrastructure for a system to exchange the limited sized messages that are now known as SMS messages or simply text messages. The packets are 190 bytes long and sport a 50 byte system overhead leaving 140 bytes for the actual message. With 7-bit ASCII (American Standard Code for Information Interchange) coding a message can be 160 characters long. For languages like Azerbaijani, German and Manx that need glyphs like 'Ü' and 'Ç' the maximum is 140 (8-bit) characters. Non-Latin languages like Arabic, Chinese and Russian are supported using 16-bit encoding, limiting the message to 70 graphemes. Users have found creative ways to cram more text into a message. Russian messages can be written using the Latin alphabet and esoteric phonetic abbreviations can be used like "l8r" for "later." For example a Dutch SMS message might read: "Xi dT FF ZZ" to convey that the sender has found the tea and is about to brew some.

Message delivery is straightforward. A new SMS message entering the system is forwarded to the nearest SMSC (Short Message Service Center). This SMSC tries to locate the roaming device. The device is either *inactive*, that is, the device is currently offline, or *active*. As long as the device is inactive, the

message is kept waiting. When the device is (or becomes) active, the message is forwarded from one SMSC to the other SMSC until it reaches its destination. A reply, whether the delivery was successful or not, is sent back to the original SMSC. An SMSC keeps trying to send an SMS messages for a limited time. In the end, it always sends back a report to the original sender of the SMS message, stating success or failure.

Nowadays, SMS centers are connected to many systems and networks like fax, E-mail, voicemail, and so on. It is feasible to send SMSes from and to a wired phone, web page, PDAs, satellite phone, and so on. To send an SMS message over the Internet, an SMSC can use a SIGTRAN (SIGnaling TRANsport) protocol extention to SS7 named SCTP (Stream Control Transmission Protocol).

Even though less restrictive cellular messaging systems have been introduced later, (i.e., EMS (Enhanced Messaging Service), MMS, and I-mail) SMS still is hugely popular. As stated above, daily, billions of SMS messages are exchanged, worldwide. Since the classical E-mail system is also a store-and-forward end-point delivery messaging system, transport of SMS messages using SS7 is a bit like transport of E-mail messages using SMTP. However, roaming is handled totally differently by the E-mail and SMS messaging systems. Below is a short comparison between support for roaming E-mail and SMS users.

Handling Roaming

Originally, the E-mail messaging system only featured end-point delivery. All messages were routed to one dedicated server, usually on a work place computer. Roaming users challenge end-point delivery routing in two ways. First, a roaming user (or device) might not be reachable at all, at times. Second, a roaming user, by definition, does not have a unique static end point. The first challenge is not a big one in a store-and-forward system. If it is impossible to forward a message, try again later. Though not optimal, this simplistic solution usually suffices. The second part, finding the current end point, is a bigger challenge. With E-mail the solution was simple, the end point was made remotely accessible, first through POP back ends, and later on through more powerful IMAP back ends. With the widespread availability of Internet at home, POP and IMAP became an integral part of the E-mail messaging system, because home computers are often disconnected from the Internet. Consequently, the current E-mail system is not a pure end-point delivery messaging system any more, from a users perspective. It could be argued, however, that E-mail is now a store-and-forward, remote-accessible end-point delivery system.

In contrast, the SMS messaging system is predominantly still is a true end-

point delivery system. This is because the end points are made to roam with the user. A cellular network, such as the GSM (Global System for Mobile Communications) network, consists of a (usually large) number of radio transmit and receive units, called *base stations*. Base stations cover a relatively small area, ranging from a part of a building to several tens of kilometers using RF (Radio Frequency) in the SHF (Super High Frequency) band and digital coding techniques. Currently the main RF channel access methods deployed by cellular networks are CDMA (Code Division Multiple Access), SDMA (Space Division Multiple Access), FDMA (Frequency Division Multiple Access) and TDMA (Time Division Multiple Access). The area where one base station has the strongest RF signal is called a cell. By definition, a cell has exactly one base station and its base station services only this one cell. Every cellular device has an identifier, called an IMSI (International Mobile Subscriber Identity) that uniquely identifies a cellular device, or more precisely, it identifies the SIM (Subscriber Identity Module) card within the device. A cellular device constantly monitors the strength of the RF signals of all the base stations it can hear. The base station with the strongest RF signal is chosen. The device notifies this base station of its presence. The base station will now register the presence of the device in two places. First, the IMSI number is registered in a local database, called the cell's VLR (Visitors Location Register), so the base station knows this device is in its cell. Second, the base station uses the IMSI number to register its cell as the current cell for this device in a systemwide database called HLR (Home Location Registers).

Effectively by this registration process, all SS7 network users can now reach any cellular device, as it roams from cell to cell. If an SMS message is sent to the cellular device, the current, or more accurately, most recent, location is looked up in the HLR database and the message is forwarded to the corresponding cell, where the cell's base station will forward it to the end point, that is, the device itself.

Naming

For cellular messaging, names take the form of telephone numbers. Traditionally, these numbers have been directly used for routing. Dialing `1234` would get a phone connected to exit four of exit three of exit two of exit one of the exchange office that the telephone was connected to. Nowadays, most telephone numbers have three basic parts: a *country code*, an *area code* (or network code), and a *subscriber number* (or mobile subscriber identity number). For local calls, the first two parts need not be explicitly provided.

Partly due to cellular phones, routing schemes needed to be adjusted and this

affected the way naming was deployed. To facilitate a fixed number for roaming cellular devices, the area code in a telephone number was used to also designate a particular cell-phone operator, effectively diverging from the geographical interpretation initially tied to area codes. This approach has nowadays been taken another step further, as the original area code is now also used to designate different types of services, such as toll-free calls, premium-rate calls, normal cell phones, pagers, and so on.

With the advent of digital telephone network technology (i.e., ISDN (Integrated Services Digital Network)), and number portability it became even harder to directly use a telephone number for routing. To remedy this, the ITU introduced the E.164 protocol in 1984.

Today *global title* naming is most commonly used. The structure of a global tile address is still hierarchical, still of variable length, but also it can contain nonnumerical values.

A global title actually is an aggregation of formats, most of which are defined in separate standards. The active format is indicated by the *numbering plan indicator*. The most commonly used numbering plan indicator values for global title routing are: `1`: for an ISDN, E.164 address, `6`: for an American, IMSI, E.212 address, and `7`: for an international E.214 address. Basically these highly structured numbers look like ordinary POTS telephone numbers to the user.

Summary of Properties

The SMS system offers delivery of a message that does not expire (once received), to receiver-specific storage, to one pre-known receiver. Though the parameters differ greatly, these properties hardly differ from the properties of the E-mail system.

1.3 Unified Messaging

The previous section described some current messaging systems and their properties, Table 1.4 summarizes them. This section introduces the notion of *unified messaging* as a candidate replacement for (almost) all current messaging.

The number of users of any given messaging system determines in a large part the worth of a messaging system to users. The more people that can be reached through a messaging system, the more usable it is to the average user. This must be the reason why new messaging systems often have at least one gateway connecting it to an older and bigger messaging system. Currently many messaging systems are connected to E-mail. For example, most cellular network

operators have some sort of gateway that allows SMS messages to be delivered to an E-mail address, and with some restrictions (SMS messages can only contain 160 characters of text) vice versa. Theoretically all messaging systems could be connected to E-mail to form an integrated set of messaging systems. Note, however, that a new unified naming convention would be needed to enable such a system. Obviously such a unified messaging system offers optimal connectivity to all its users. This might be the very reason why so many messaging systems have been connected to E-mail. The current situation is illustrated in Figure 1.7 which shows two interfaces (a personal computer and a cell phone) and a small selection of current messaging system protocols. There seems to be an implicit consent that unified messaging is feasible and usable. It is this implicit consent that prompted the research for this dissertation. There are a number of questions that should be asked and thoroughly researched. These questions are:

Is (near) unified messaging feasible?

Would unified messaging be usable?

What infrastructure would support it?

The short answers are: "yes," "yes," and "micro-objects."

Feasibility of Unified Messaging

If all messaging systems could be integrated by connecting them to E-mail, then unified messaging would clearly be feasible. Although E-mail provides the proper means to transfer messages, one could argue that there are inherent

System	Properties
E-mail	Permanent, receiver-specific, private.
USENET	Impermanent, class-specific, public.
IM	Transient, group-specific, shared.
Weblog	Permanent, sender-specific, public.
SMS	Permanent, receiver-specific, private.

Table 1.4: Summary of the messaging systems and their properties as discussed in Section 1.2.

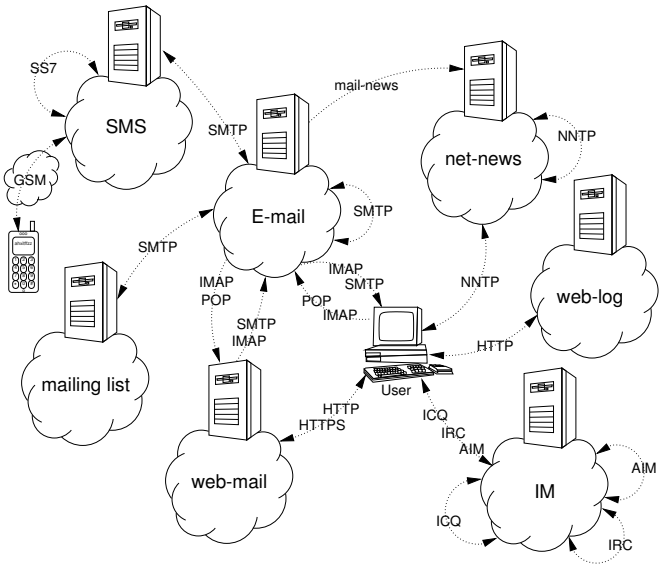

Figure 1.7: A few interfaces and messaging system protocols. Note that a smartphone can take the role of both a phone and a computer.

problems with integration through E-mail. The properties discussed in Section 1.2 of USENET and IM differ considerably from the properties of E-mail. Besides, IM has synchronous properties, and USENET uses flooding to gain high accessibility. It is not obvious how E-mail can be used to support these messaging systems. So E-mail cannot fully play a unifying role. However, in Chapter 4 a detailed description of a unified messaging system is given, thus asserting a positive answer to the feasibility question.

Usability of Unified Messaging

At first the usability question seems a nonissue; Every user can send a message to any number of other users. Actually since E-mail has so many users and it has been usable for a long time, it is probable that a system that is only an order of magnitude larger could be usable too. However, as any long time E-mail user will probably agree to, E-mail seems very prone to spam or unsolicitated messages, also referred to as, *junk-mail*, UCE (Unsolicited Commercial Email), and UBE (Unsolicited Bulk Email). The vast majority of spam receivers dislike it as we will return to in Section 4.3. However disliked, low costs combined with huge audiences make it very cost effective for the spammer, even if the re-

sponse is ten per million. As stated above, about 90% of all E-mail messages are spam. Also according to the Radicati Group, a typical 1,000-user organization can spend upwards of $1.8 million a year to manage spam. Spam for E-mail and USENET together with spim (spam over IM), could become a threat to the usefulness of the Internet as a whole. Figures released by Microsoft at the October 2010 RSA conference indicate that 87% of all spam messages find their origin in some botnet (roBOT NETwork). A botnet is a collection of malware infected computers on the Internet. The malware secretly takes orders from one or more controller computers. A botnet controller could order an infected computer (or "bot") to generate and send spam.

Microsoft reports that, between March and June of 2010, one single botnet controlled from Lithuania, was responsible for 56.7% of all botnet-sourced junk mail. The Rustock and Cutwail botnets were the next two most prolific junk mail sources in Q2 2010, churning out 16.9% and 15.4% respectively. According to Jeremy Kirk from IDG News Service on August 2010, Symantec's MessageLabs' division came to a similar conclusion. So a unified messaging system as large as the E-mail, the USENET and SMS messaging systems united might collapse under the spam load.

There are at least three ways to combat spam. One, make it expensive; two, use automated spam filters; three, put access control in the hands of the receiver. Since it is very hard to make spam expensive without some form of accounting, the first solution might add complexity and cost for all users. Also the accounting might put too much power in the hands of the accounter. Automated spam filters make mistakes, and not every user will use them to the fullest. Hence not all spam can be filtered out and spam might still be cost effective. Worse, since filtering happens at or near the destination it only battles delivery, not overhead. Worse still, filtering might lead to more spam being sent in an effort to get through. The third solution tackles the problem at its root. Spam is a problem only if the senders can send messages to whomever they want. Spam would be extremely restricted if receivers can predetermine from whom they want to receive messages. This solution, however, necessitates a paradigm shift, because users will have to get used to the concept and management of predetermination. A more detailed look at spam is presented in Section 4.3.

Infrastructure for Unified Messaging

Many messaging systems run directly on top of the Internet. More and more telecom messaging systems are migrating to the Internet too, relying on telecom networks, only for the last mile to the user device. So clearly the Internet is capable of supporting many conceivable messaging systems, including the

integration of a number of key messaging systems, and thus by extension, including a unified messaging system. That does not mean that it is simple to construct a unified messaging system (as described Section 2.2) on top of the Internet protocol stack.

As part of our research, a simple proof-of-concept implementation was written, in Java, directly on top of the Internet protocol stack. Extending and maintaining this relatively simple program was increasingly difficult as features were added. As it turned out, the unified messaging system, by most definitions, could be considered a large-scale, distributed application. As such the unified messaging system met the same difficulties that have been noticed independently by several groups researching distributed systems, as will be shown in Section 3.1. The distributed aspect of the unified messaging system was, in fact, generic enough to be captured in a separate layer, on top of which the unified messaging could be constructed. After researching current middleware for distributed systems, no middleware layer seemed suitable for the unified messaging system. Further research led to the design of a novel distributed programming middleware model, that was better suited for a specific class of distributed applications, including unified messaging. This middleware layer turned into a separate research topic, as described in Chapter 3.

1.4 Messaging System Taxonomy

The section above is lacking a formal definition of what properties a unified messaging system should have. In order to give a formal description of any messaging system, a classification scheme is needed. This section describes such a classification or taxonomy for messaging systems and the formal definition of a unified messaging system is postponed until the next chapter. The taxonomy is organized along the four most important aspects of messaging systems from a *user's perspective*, as opposed to a technical or design perspective. With this taxonomy, any messaging system can be scaled with respect to four independent dimensions. Figure 1.8 shows the four dimensions and their values.

Dimension 1: Time

A messaging system can have one of three values in the time dimension: *immediate*, meaning that all messages are transient, short-lived or available only once during a relatively short period; *impermanent*, meaning that all messages are available pending their expiration or revocation by some set of rules; *permanent*, meaning that all messages are available indefinitely unless a message is

Dimension	Values
Time	Immediate, impermanent, permanent.
Direction	Simplex, duplex.
Audience	World, group.
Address	Single, list, all.

Figure 1.8: The messaging system taxonomy.

explicitly revoked by an authorized user.

Dimension 2: Direction

A messaging system can have one of two values in the direction dimension: *simplex*, meaning that a write-only storage or channel is used for message delivery, a reply has to be directed towards another storage or channel; *duplex*, meaning that one store or channel is used for both reading and writing.

Dimension 3: Audience

The audience of a messaging system is the set of users that *can* receive a message through this system. In the audience dimension a messaging system can have two values: *world*, standing for every user that has the hardware, software, and connectivity to use the system or *group*, standing for a true subset of all users. In a grouped messaging system, users cannot post messages to a user outside their audience, even though this outsider is ready for any message and uses the same system. Restriction of audience (grouping) can be the result of restrictions related to the infrastructure or implementation. The system can also limit the audience as a service, security measure, or due to specific policies.

Dimension 4: Address

In the address dimension a messaging system can have three values: *single*, if the system allows only one recipient per message; *list*, if the system allows for addressing more than one explicitly addressed recipient; *all*, if the system allows for some form of broadcasting.

Classifying Messaging Systems

The four dimensions are truly independent, although not all of the thirty six types of possible messaging systems might be equally useful. When classifying a messaging system, it is easy to confuse audience and address: both are subsets of recipients. The audience comes with the system and users have no direct influence on it. The address is something the user determines (per message) and the system has no influence on. The intersection of audience and address is the set of recipients that is supposed to receive the message.

Figure 1.9 shows the classification of the five messaging systems as described in the previous section (Section 1.2). Many interesting observations can be

System	Time	Direction	Audience	Address
E-mail	Permanent	Simplex	World	List
News	Impermanent	Duplex	Group	All
IM	Immediate	Duplex	Group	All
Weblog	Permanent	Duplex	World	All
SMS	Permanent	Simplex	World	Single

Figure 1.9: Classification of some current messaging systems.

made when using the taxonomy to compare messaging models. For example, the fax messaging system and the SMS messaging system could both be classified as (immediate, simplex, world, single) systems, revealing that they share an underlying model. Due to the different output devices and infrastructure, their systems are, however, very different and incompatible. Interestingly but not surprisingly the SMS system is meeting exactly the same user demands for service extension that the fax system (invented over 100 years earlier) has met. Users will want support for sending a single message to multiple recipients, automatic forwarding to other recipients or locations, nonrepudiation, authentication, and so on. Note that voicemail systems are also (immediate, simplex, world, single) and were also confronted with similar user demands. Researching the history of several messaging systems, one might conjecture that messaging systems with coinciding positions in the taxonomy, usually have coinciding development paths. However, this is outside the scope of this thesis. Another interesting observation is that when a system is built on top of another system, its classification clearly reveals what property (if any) has been downgraded in

favor of the upgrading of some other property. For example, due to some limitations of E-mail [69], several so-called *mailing-list* systems were built on top of it. With a mailing-list, users would be able to send an E-mail to a symbolic E-mail user. This message would then be forwarded automatically (by a mailing-list server) to a set of users. Sending special E-mail messages, for example, a message with the subject "subscribe," would allow users to be included in the set of recipients. The mailing-list messaging system can be classified as (permanent, duplex, group, all). Comparing this to the classification of E-mail to the mailing-list messaging system, it is clear that audience was downgraded in favor of upgrading its addressing capabilities.

Armed with this taxonomy, a formal description of a proposed unified messaging system will be given in Section 2.2.

1.5 Diffusion of Unified Messaging

Next to research questions about unified messaging, one practical question on this subject manifests itself. With unified messaging, feasible, usable, and implementable, will it have practical value? Since a mass-messaging application is true to its name only if its usage is ubiquitous, this question translates into this; "Will unified messaging diffuse, and if so, how fast?"

The nonscientific answer to this nonpragmatic question probably is "No." Due to the cesspool of intermixed financial and other business interests in messaging, it is highly unlikely that at some point in time all major parties will want to standardize.

Probably if everybody with a communication device, be it cellular or not, would pay a fixed amount, of say US $ 5, per month for communication service, regardless of usage, there would be more than enough money to support the service. This type of payment is often referred to as *flat rate* (or sometimes as linear rate because the rate is linear with time). Flat rate would go well with a unified messaging system. It would make it everybody's interest to bring cost down, and clearly unification of the core systems would be a good first step. However, in the current situation, it is often most profitable to offer yet an other way of communicating, because the vast majority of communication still is billed per message.

Hopefully, however, this dissertation, or its research, will play some small role in the unification process.

1.6 Social Network Messaging

The last ten years, a number of online social networks grew past one million users. Examples are Facebook, LinkedIn, MySpace, and Twitter. An argument could be made to include YouTube, eBay, and dating sites like Zoosk. These social networks could be qualified as multi-author weblogs. In general, this is not how they are perceived by their users. From a technical point of view, however, Facebook (in its 2012 incarnation) is a multi-author weblog with integrated instant messaging and E-mail. Twitter is like a micro-blog site with an SMS interface. Most of these online social networks are a mix and match of messaging functionality. As such they feature partial unified messaging. Since unifying messaging systems is not their goal and since they do not offer unique messaging functionality they play a minor role in this thesis.

1.7 Summary

This chapter described some mass-messaging systems and introduced a taxonomy to classify and compare these and similar systems. It introduced the concept of unified messaging as a valuable research topic. The next chapter will introduce the unified messaging model, as designed by the author and his promoter, and discuss the major differences between the unified messaging model and current messaging practices.

❧ *"Can a single messaging system cover the whole taxonomy?"* This question seemed rhetorical. There was no doubt in my mind that I could design and build one. The doubt came later, after I tried to build a proof of concept. The first attempt, in PHP, failed and I told my promoter *"PHP is unsuitable."* The second attempt, in Java, failed and I concluded that *"The Enterprise JavaBeans architecture is too complex."* The third attempt, in C, totally collapsed under the complexity of distributed computing. *"C is too simple,"* I remember lamenting. On the positive side, every attempt simplified my design, ending up with a minimalistic one that had only three major concepts: messages, targets, and access control. With those three I could mimic any type of messaging within the taxonomy. It was good to have failed three times in a row, as it helped shape the solid design for a unified messaging system. It was also bad to have failed to implement even the simplest proof of concept, as it made me worry about the feasibility. My promoter told me he shared my worries. *"It is probably impossible, but if it were not…."* The rest of the question would become the next topic of research: *"What would it take to implement a unified messaging system?"*

Chapter 2

A Unified Messaging Model

> "The long-term goal is to get E-mail available on every Treo with every [BlackBerry, Seven, Visto, and GoodLink] server."
>
> *Joe Fabris*, the director Palm wireless marketing.

The taxonomy presented in Section 1.4 allows analysis and classification of messaging models. Moreover, this taxonomy dictates the services that *unified messaging*, as defined in Section 1.3, needs to provide to the user. Throughout this chapter, the word "user" can be read as "the application program on behalf of the user."

A unifying messaging system must, by definition, allow the user to choose the properties of a message exchange, on a per message basis. In terms of our taxonomy, this translates into the user's ability to freely determine the position of every message exchange in the messaging space spanned by the four axes of the taxonomy. In other words, a unified messaging system has to be adaptable to the changing messaging needs of the user.

A common definition of "adaptability" is "the ability to change (or be changed) to fit changed circumstances," (see [21]). In the context of a taxonomy, "change" can be interpreted as a change of position, leading to the following definition:

> *A system has "maximum adaptability" within a given taxonomy, if the system can easily move or be moved to any position within that taxonomy.*

If a messaging system with maximum adaptability already existed, it could replace most of the other messaging systems; there would be only one big messaging system and some nongeneric systems would be filling the remaining niches.

Since there exist many different messaging systems each with their own merits and peculiarities, one might assume none of them has maximum adaptability. Using our taxonomy to categorize existing messaging systems, it is clear that this assumption holds: none of the massive-messaging systems has maximum adaptability; most such messaging systems lack adaptability altogether. This chapter introduces a *unified messaging model* that does have maximum adaptability and could be used as a basis to unify, incorporate and replace all major existing messaging systems, including WhatsApp, E-mail, fax, SMS, IM, I-mail, USENET, weblogs, MMS, voicemail, and so on.

This model illustrates the feasibility of a large-scale UMS (Unified Messaging System) that supports maximum adaptability, and which is capable of providing the same services as existing messaging systems. To the best of my knowledge such a messaging systems does not yet exist. Section 2.4 contains some examples how this model could be used to closely mimic existing messaging systems. Besides existing messaging systems, some examples demonstrate messaging systems with a hitherto unknown mix of properties.

2.1 The Unified Messaging System

Before describing the model for the UMS, an informal description of a UMS is in order. Basically, the UMS is a middleware layer that manages the distribution of message objects. On top of the UMS middleware layer sits the application layer, providing a (graphical) user interface or proxy. The UMS does not have, or even need, a name-space service, but several name-space layers can operate next to the UMS layer. To offer compatibility with current messaging systems, the application layer could use a gateway layer to one or more existing messaging systems, as shown in Figure 2.1.

2.1.1 Missing Services

Next to the messaging service provided by the UMS, there are some services missing that are provided by some of the existing messaging systems. At first, it might seem that by definition, these services should be part of the service provided by the UMS. However, after some research, these additional services have been proven to be unrelated to messaging, or to be only cosmetically different from the messaging service provided by the UMS. Some of the seemingly most blatant omissions are discussed below.

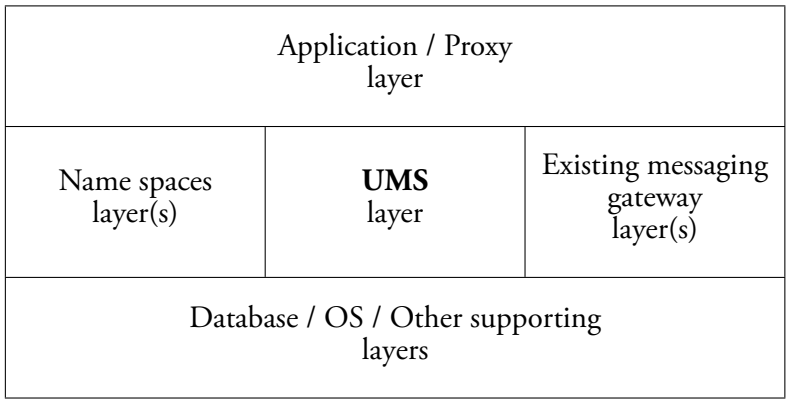

Figure 2.1: The position of the UMS-layer.

No Presence Service

The UMS does not include presence service because presence basically is a specific form of messaging. Presence can be implemented by sending a "I am alive" message at fixed time intervals. Nongeneric messaging systems might necessitate a separate implementation of presence messaging. A truly generic messaging system should be able to handle presence as just one of the many messaging needs of the user.

No Security Service

The UMS does not include secrecy, authentication, nonrepudiation, and integrity control services, because these should be done *end-to-end* (i.e., at the application level) [53]. Although the UMS layer does employ encryption that might be enough for some uses, it would be impossible to facilitate (and maintain) generic security that would suit all users, all the time. Note that there are ways to provide PGP (Pretty Good Privacy) by adding end-to-end security to messaging systems like E-mail (see [9]). Chapter 3 introduces a library component that will take care of this.

No Name-Space Service

A name-space service maps a user-friendly name to machine-usable addresses. Mapping has two distinct advantages. The first advantage is that the user is allowed to use a human readable name instead of a hard to use machine address, the second is that the user is shielded from address changes. The first advantage

can be exemplified by considering the difference in effort it takes the average human to memorize "cnn.com" and "157.166.226.25" For an example of the second advantage, consider a server moving to a new machine address, rendering its old address useless. With name-to-address mapping, the name could be made to map to the new address, allowing references-by-name to continue working.

A well known example of a name-space system is the DNS, as described in the section on E-mail (Section 1.2.1). DNS maps several name spaces into the IP address space. Well known name spaces are E-mail host names (MX records) and DNS server names (NS records). Another well known example is the name-location to telephone-number mapping in phone books. The phone book name space features a distinct third advantage for users; it is searchable.

Usually a dedicated name space is integrated into, or closely bound to, the system using it. There are good reasons for the UMS to not have an integrated name space. The main reason is that such a name space would need to unify all the properties of all other name-space systems which in turn would lead to a *generic* systemwide searchable name space, which would need a complex access control mechanism or a pricing mechanism, to combat spam. The UMS does not need an integrated name-space service. Because the three advantages stated above, can be achieved if need be, by combining the UMS with one or more existing name spaces. The advantage of friendly naming can be handled by one or more nonintegrated name spaces. The first and third advantage, user-friendly names and searchability can be served by any number of nondedicated name spaces. As will become apparent from the rest of this chapter, the UMS is a transport service, where messages are kept alive long enough allowing messages to reach their destination user application layers comfortably. Therefore information is usually not very long lived within the network, days rather than weeks, lessening the effects of address changes. So the second advantage, machine-address independence is usually not (fully) needed. However, if some information has to be long lived, an existing name-space system could be used. In the latter case, spam becomes more expensive, because the sender will have to provide the resources. The subject of spam will be dealt with in detail in Section 4.3.

No Delete-Message Service

The unified messaging model does also not provide a "delete" service. This is in line with most other messaging systems. In fact, messages are immutable. Releasing a message to a messaging system indicates that the message is finished. Once a message is sent, it cannot be un-sent. Note that some messaging systems

feature a "remove" operator. The actual removal of a copy is, however, left to the server(s) holding it, there is no guarantee that all copies will be removed by the servers holding a copy. Also there is no way to revoke a message once a recipient has read and or copied it. For example, USENET has a highly insecure "cancel" command to delete messages already posted. Only the NNTP server of origin is supposed to cancel a message. The cancel command is seldom used by users, it is, however, (ab)used by cancel bots, that is programs that attempt to delete spam. Due to the voluntary nature of these delete operators, it can be easily shoehorned on top of any messaging system by means of a special message. Such a cancel message basically is a request to remove an earlier message. Since such a cancel message can be a normal message with a special content, such a voluntary removal scheme can be implemented at the application level, should the need for it ever arise.

One consequence of the lack of a delete operator is that messages might live forever. With massive messaging, this can cause a storage problem. Actually, all of the better known messaging systems use one form of expiration or the other. For example, USENET articles expire after several days or weeks and E-mail messages are (supposed to be) deleted after each successful hop, leaving only one private copy at the end point, out of reach of the local MTA. The unified messaging model also supports message expiration but it is up to the application layer to choose the expiration date of outgoing messages, as well as to extend a message's lifetime, if need be.

2.2 Unified Messaging Model

After the terse introduction above describing the unified messaging system, this section will elaborate on its underlying model.

2.2.1 Goals

One of the prerequisites of a model for a UMS is the maximum adaptability. Clearly this is not the only property a unified messaging model should have. The unified messaging model is designed with the following goals in mind:

1. *Large-scale messaging*: handling hundreds of billions of messages per day between billions of users.
2. *Independence of trusted sites*: allowing (a combination of) a client/server or a peer-to-peer communication model [46].
3. *Prevention of spam*: preventing unsolicited messages, without restricting the freedom of speech.

4. *Orthogonality of dimensions*: deciding on time, direction, audience, and address independently.

Note that the goals of an *implementation* are a superset of these goals. In Chapter 4 the goals of the UMS *implementation* will be enumerated. They will include ease of use, efficiency, and maintainability among others. For the *model* such goals are an undesirable burden.

2.2.2 Target and TISM

There are some things that, by my definition, all messaging systems have in common. Every messaging system has some digital representation of a message. Also every messaging system has some kind of storage unit for those messages. Messaging systems typically have their own nomenclature. A message can be termed "mail," "text," "post," "tweet," "message," etc. The same naming diversion can be found with regard to the place where these messages can be found. Typical names are "mailbox," "channel," "bin," and "group."

A UMS needs some new terminology to counter possible assumptions due to the reuse of known nomenclature. In the UMS each message is "targeted" because it is directed towards a specific user or a group of users, a message is immutable because it cannot be changed after it has been sent and it is usually short. From this reasoning the acronym TISM (Targeted Immutable Short Message) has arisen. Also, the word "target" is used to denote the destination of a TISM. Within the model, "messaging service" is defined as making a TISM accessible for a group of users, by posting it to one or more targets. It will come as no surprise that the main objects in the unified messaging model are: *TISM*, containing a subject and a message body; and *target*, containing a set of TISMs.

2.2.3 Protection and Identification

A target protects each TISM with public key encryption and a message digest [55]. In our model each target is associated with a unique *post-key*/*read-key* pair. To post a TISM, the proper post-key is needed. Likewise, to read a TISM, the proper read-key is needed. Without a read-key, it is sufficiently hard to reconstruct a TISM, even if a post-key and a copy of the encrypted TISM are available. Without a post-key, it is very hard to spoof a TISM even if the read-key is available. The UMS will generate a new post-key/read-key pair for every new target.

In the UMS a target is identified by a systemwide unique bit string. We define a *target-ID* as this unique bit string. To access a target, the proper key is

needed in combination with the target-ID. This combination is called a UMS tuple. A UMS tuple consists of one or two keys and a target-ID. For convenience a (post-key, read-key, target-ID) tuple is denoted as a *post/read-tuple*. Likewise, we use (read-key, target-ID) as a *read-tuple* and (post-key, target-ID) as a *post-tuple.* When the UMS creates a target for a user, the user is returned a post/read-tuple, from which a separate post-tuple and read-tuple can be created. Typically, a user might create a target and distribute its read-tuple to others, enabling them to get the encoded TISMs from the target (using the target-ID) and to decode those TISMs (with the read-key). This is similar to a weblog-messaging system. Had the user distributed the post-tuple, an E-mail like messaging system would have resulted, as it would allow people to post messages for that user to read.

The UMS user has a number of ways to distribute UMS tuples. For example, a user could pass on a tuple wrapped in a TISM, distribute a tuple through the World Wide Web, or store a tuple in one of more name-space systems. Other lookup models are also feasible. Note that not being bound to any particular lookup mechanism or name-space service is one of the strengths of the UMS.

2.2.4 Taking Control

To utilize the fine-grained control the UMS offers, the user needs a separate target for each different communication partner or group. This may sound complex, especially to users that manage all their Internet E-mail from one so-called, in-box. However, most E-mail users already have many sub-mailboxes. Likewise, most IM systems allow users to create any channel/room they want to. As another example, every USENET user can create a new `alt.*` group at will (like the actually existing `alt.swedish.chef.bork.bork.bork`). Creating a new box or channel, in one of these legacy messaging systems, is limited by the ability to create a new entry in the accompanying name space. For example, finding a meaningful name for a new `alt.*` USENET group that does not already exist, is hard, as is the case for IM channels/rooms.

In the UMS system, the target-ID is not bound to any name so users can easily create *thousands* of targets if need be. Note that this will necessitate support from the application layer to hide complexity.

2.3 Resource Allocation

One thing that is radically different in the unified messaging model when compared to legacy approaches is how and when resources are being used. Most

messaging systems push their messages as far as possible towards the receiver. Clearly, this makes sense from a performance point of view. However, it also allows spammers (i.e., users that send out spam), to send out vast numbers of messages (typically resulting in millions of copies being distributed), in a short time. Messages are simply pushed into the messaging system, and after that resources of other participants (i.e., likely receivers) are being used for delivery, resulting in shorter delays for the receivers. With modern networks, performance is not so much of an issue any more. Checking a remote message store every minute for changes, is no problem for most current network infrastructures. This makes *message-pull* systems a viable alternative. The basic distribution method of the unified messaging model is message-pull. Every target and TISM has a *home location*. This is the place of origin in the form of a server on the user's network. It is also the only place that has to hold on to the created target or TISM. There are two important observations to make about this. First, unlike most messaging systems, the poster has to supply the (storage, process and bandwidth) resources. Second, there is at least one known place where the message is available (until it expires). This illustrates how the UMS puts the recipient in control, in contrast to most existing messaging systems.

Note that the unified messaging model does allow consenting parties to use message-push or other replication policies on top of this basic distribution. This is facilitated by additional replication at a lower layer as described in Section 3.5.6

2.4 Mimicking Legacy Messaging System

Since the unified messaging model should be able to accommodate virtually all other messaging systems, I will give a coarse description of UMS-based applications that mimic the functionality of the following legacy messaging systems:

1. *Internet E-mail*: being large and well known.
2. *USENET News*: targeting groups of users.
3. *Instant messaging*: featuring a real-time component.
4. *Web logging*: featuring subtle rules for posting.

The functionality of these systems is defined in accordance with the taxonomy as discussed in Section 1.4.

2.4.1 Internet E-mail Imitation

The E-mail system is a (permanent, simplex, world, list) messaging system. A message-management program (let's call it u-mail) would distribute a post-tuple

of a newly created target (let's call it mailbox target). U-mail would further display selected TISMs from the mailbox target. U-mail would allow new TISMs to be posted to any target it holds a post-tuple for.

To be backward compatible with the Internet E-mail system, the u-mail program could also feature legacy protocols like SMTP, POP3, and IMAP4.

2.4.2 USENET Imitation

The USENET News system is a (impermanent, duplex, group, all) messaging system. A message-management program (let's call it u-news) would allow users to create a new target and distribute its post/read-tuple. The u-news program would list the short text of the TISMs of the subscribed targets. Users can then select the TISMs they want to read. U-news would further allow users to read from and post to those subscribed targets. A fair amount of backward compatibility could be realized here too. Section 2.5 will show a way of distributing UMS tuples to mimic newsgroup moderation.

2.4.3 Instant Messaging Imitation

Most IM systems (like WhatsApp, BBM (BlackBerry Messenger), AIM (AOL Instant Messenger), ICQ, Skype Chat, and IRC) are (immediate, duplex, group, all) messaging systems [25]. An IM interface program (let's call it u-talk) would allow users to create a target and distribute its post/read-tuple. U-talk would allow users to select a target. The u-talk program would supply the selected target with a call-back function, that the target would call upon the arrival of new TISMs. The u-talk program would display a split screen, showing all the new TISMs in the top half and allow the user to type in lines of text in the bottom half. The u-talk program would post each line the user types, prefixed with a user alias, as a new TISM. Again backward compatibility could be introduced.

2.4.4 Web Logging Imitation

A weblog is a combination of simplex and duplex communication. In its purest form a blogger (i.e., a user who runs the weblog) appends messages to a page on the WWW. The blogger can append messages and other users can only read the messages. However, users can react to a weblog message by posting a follow-up message. Weblogs thus form a (impermanent, simplex/duplex, world, all) messaging system.

A weblog management program (let's call it u-blog) would allow a user to create a new target (let's call it blog target) and distribute its read-tuple. U-

blog would further allow selecting a blog target and reading TISMs from the selected blog target. U-blog would allow a blogger (i.e., a user in possession of a post/read-tuple) to post a new TISM to a blog target. Appended to this TISM is a post/read-tuple of a newly created target (let's call it follow-up target). U-blog would allow reading from, and posting to, any follow-up target. Obvious extensions, like having multiple bloggers or moderated follow-ups can be realized along similar lines, using still more targets and careful management of the UMS tuples.

2.5 New Messaging Paradigms

As we have shown, the UMS can be used to mimic existing messaging systems, but the model has much more to offer. Proper distribution of UMS tuples allows the UMS to implement almost any imaginable messaging system. Some might find it surprising that with only a few tuples, some novel and complex forms of messaging can be facilitated. The (a) Figures from 2.2 through 2.7 show possible messaging system. Tuples are depicted as solid arrows showing the allowed TISM flow. A thicker solid arrow signifies ownership. The (b) Figures from 2.2 through 2.7 show a corresponding example usage. The numbered dotted arrows show the path an example TISM would travel from poster to reader. In all figures the application layer program is depicted as a stick-person and the target is depicted as an amoeba-like blob.

If there would be one target for which all users had a post/read-tuple, it would be a form of a say-all, hear-all messaging system (see Figure 2.2). It would be hard to deny access to individual users. Note that in the UMS model, it is feasible to have a high-volume target like this without a huge central server, because TISMs are stored at the poster's home location. By limiting others

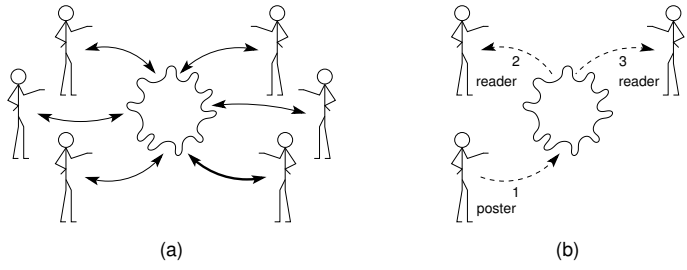

Figure 2.2: Say-All, Hear-All Messaging. Note how hard it would be to ban an individual user in possession of a post/read tuple.

to read-only access, the say-all, hear-all messaging turns into publish-subscribe

messaging. If one user (called the publisher) created a new target and publicly announced a read-tuple, a simple form of publish-subscribe messaging could take place. Figure 2.3 shows just that situation, with one publisher and three subscribers. This model can be extended even further. If one user (called the

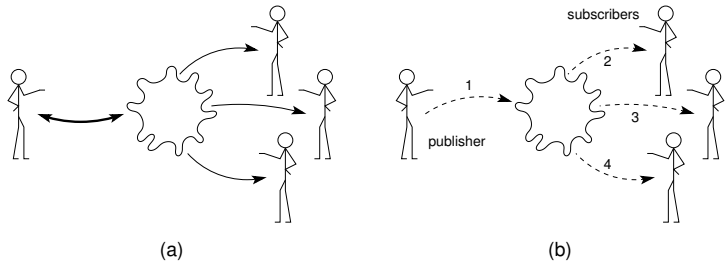

Figure 2.3: Publish-subscribe Messaging. Note how hard it would be to ban an individual user in possession of a post/read tuple.

moderator) created a new target (i.e., a moderated target) and distributed a read-tuple to a number of other users, a form of moderated messaging would result (as shown in Figure 2.4). The moderator would cross-post a selection from the unmoderated target into the moderated target. The beauty of this scheme is that any user can start doing this at any time. By the way, even if the moderated target would contain many TISMs, the required resources for the moderator would be modest for there is no need to copy all the TISMs from the unmoderated target, only the meta information would have to be stored. Some-

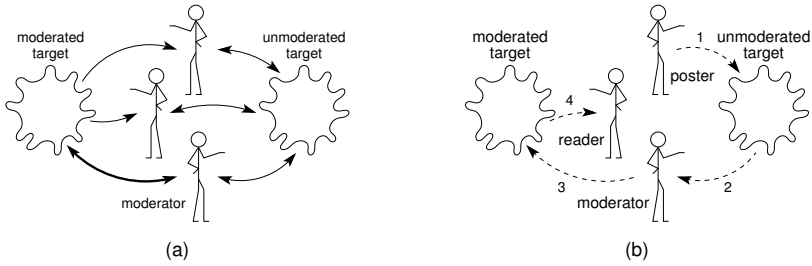

Figure 2.4: Moderated Messaging. Note how the moderator can decide on a per TISM basis to forward a message or not. The moderator however cannot ban an individual poster.

times a moderator needs the ability to deny access to an individual user (let's call him BIFF [56]). A messaging system can be constructed such that input from BIFF can be made invisible. First, the moderator creates an input-moderated target and distributes the read-tuple, just as in the previous example. Then, the

moderator solicits from each user the read-tuple for a new target. From these so-called, input targets TISMs can be cross-posted into the moderated target. The result is shown in Figure 2.5. This way a moderator can oust BIFF simply by ignoring BIFF's input target. Reverse all the arrows in Figure 2.5 and an output-moderated messaging system would result, allowing the moderator to select none, or a number of appropriate TISMs for each individual user. A combination of both would lead to a highly configurable input-/output-moderated or total-moderated messaging system. Total moderation is, in fact, a combina-

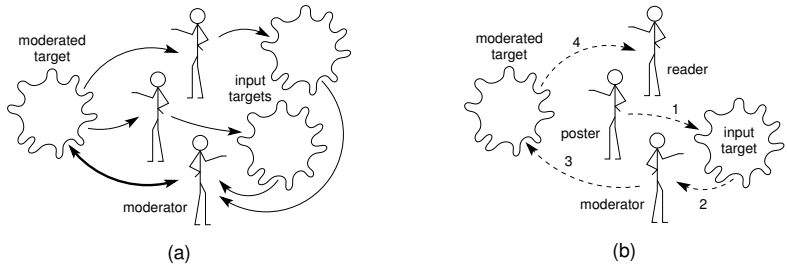

Figure 2.5: Input Moderation. Note how, contrary to Figure 2.4 the moderator can now also ban an individual poster just by ignoring their input target.

tion of manual selection and manual forwarding. The forwarding part could be done automatically by a computer (i.e., a forwarding server), to a given set of users. Automatic forwarding is best known under the name of "mailing list." A mailing list is a form of read-access control. Note that it is easy to confuse moderating and access control. Moderating is done on a per-message basis, and access control is per-user based. Besides read-access control, there is, of course, post-access control. Access controlled posting can be seen in the form of the registering mechanism, as featured by some weblogs and IM systems. Currently there are no well known messaging systems that offer full read- and post-access control. The closest thing to a messaging system with full-blown access control would be a moderated E-mail mailing list or a moderated BBS (Bulletin Board System). However, any user in possession of the address of a mailing list can post messages to the moderator, even if the user is not on the list. These kind of "holes" in control exist because the control was shoehorned on an existing system. Holes like these can be plugged by using special mailing list or BBS server software like Fluxbox, Dada Mail, UseBB, or Mailman.

In the messaging system as shown in Figure 2.6, it is possible to prevent posting by an individual user. In this messaging system the automatic forwarding A or B could be stopped. For a picture of a classical (i.e., output controlled)

mailing list messaging system, we need to simply reverse the arrows.

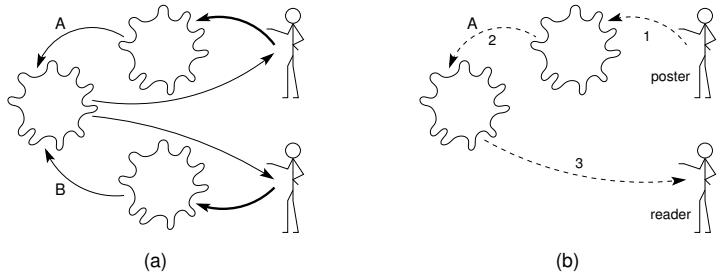

Figure 2.6: Input Controlled Mailing List. Note that this messaging model is the same as the one in Figure 2.5 with the moderator replaced by automatic forwarding processes A and B allowing per poster moderation, though not of an individual reader.

A fully controlled mailing-list like messaging system is depicted in Figure 2.7. By controlling the automatic forwarding A, B, C, and D, it becomes possible to control which individual user can post and/or read.

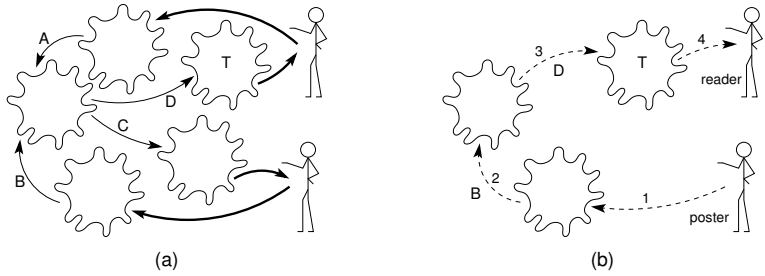

Figure 2.7: Fully Controlled Mailing List. Note how in this messaging model automatic forwarding A through D can be cut to regulate posting and reading on a per user basis.

It can be hard to discontinue a subscription to an Internet E-mail mailing list, but it is always simple to unsubscribe from a UMS mailing list, because in this system, a user does not depend on another person to stop the forwarding. The user, in Figure 2.7 who is reading from target T can stop reading from it at any time, and because target T collects TISMs *only* from this specific mailing list, no other TISMs are lost.

This is a general feature of the UMS model. Users can be very selective, since each communication platform is supported by an ad hoc created messaging system. Users will have many targets to receive TISMs from: their spouse,

their boss, their mother, their company's mailing list, their hobby club, their government, and so on, each of which can be ignored independently without further ado. As mentioned above, the poster is initially responsible for resources, not the receiver.

We have shown, with these few examples, that careful creation of targets and distributing their UMS tuples, can lead to the creation of sophisticated messaging systems.

2.6 Summary

This chapter described a basic model for massive messaging that incorporates all the current messaging systems, as well as any messaging system that fits the messaging system taxonomy. I have introduced the concept of target and TISM, and discussed the major differences between the unified messaging model and current messaging practices. The next chapter will in detail define, describe, and discuss a split off of a generic distributed-object layer from the UMS layer.

❧ *"What would it take to implement a unified messaging system?"* My first thought was *"a miracle."* My promotor pointed out that miracles are often misunderstood normalities, *"just keep dividing miracles in smaller miracles until they seem normalities."* I realized he was talking about software abstractions and after some pondering I found three simplifying abstractions. First, *content-centric addressing* for location-agnostic access to data objects (e.g., TISMs), second, *clustering* for data-object grouping (e.g., targets), third, *replication policies* for meta information (e.g., TISMs added to targets). I remember thinking *"three miraclettes"* because it is like sitting atop a cloud from where requested objects magically appear, in a whimsical structure imposed upon the cloud and soft whispers telling all about that cloud. My promotor, however, was unperturbed: *"Implement these abstractions first, it will simplify the rest."* I realized that this might be doable if the distributed data objects were dumb and small. If done well, these "micro-objects" would make perfect building blocks for a whole class of distributed application objects, not just targets and TISMs. It would leave only an easier question: *"How to implement unified messaging atop a cloud of micro-objects?"*

Chapter 3

Micro-Objects

> "People should learn how to play Lego with their minds."
>
> *Vitorino Ramos*, bio-inspired computation specialist.

Implementing a middleware layer to provide a unified messaging service can be quite challenging. A naive approach would be to translate the concepts of the unified messaging model, one-on-one, to application objects. The proof-of-concept implementations that were initially written as part of this research had a target and a TISM class. Defining the interface for these classes was almost trivial, however, implementing these classes proved to be challenging. Most challenges had to do with the large-scale distributed nature of the system. To facilitate the design and implementation the middleware layer was split into two layers. The bottom layer of the stack would isolate the distribution part, and top one would implement the messaging service. The idea was to use an off-the-shelf, or slightly adapted, implementation for the distribution part. Selecting an existing implementation proved hard. All implementations reviewed had some drawbacks. There were, however, problems that none of the candidates solved, notably those related to partial failure. An important lesson from these selection efforts was one of the fundamental truths of distributed systems:

> *Partial failure cannot be made transparent, rather, it should be made apparent and confined to a single distribution layer.*

Partial failures happen when one information exchange, in a related series of several, fails. For example, Alice sends some information to Bob. If no timely

confirmation is received, either it is lost, still forthcoming, or Alice's initial information has not been received (yet). As the parties do not know whether their actions are based on the same information, this poses a coördination problem. With large-scale distribution, there are many, not just two, parties second guessing each other, and the problem is aggravated. It turns out that the more complex the shared data is, the more complex it becomes to deal with partial failures. In this chapter a novel design is presented for the lower layer of our implementation and that does minimize the problems with partial failures.

3.1 Distribution Woes

Creating a truly large-scale distributed application has since long been recognized to be a nontrivial exercise despite the existence of various distributed-application development frameworks. Astley et al. [8] notice that the asynchronous nature of distributed systems significantly complicates application development. Klintskog et al. [42] notice that transparency is possible, modulo failure, timings, and resource consumption. Frameworks like CORBA (Common Object Request Broker Architecture), .NET Remoting, and EJB (Enterprise Java Beans) try to map the object-oriented programming paradigm to networked environments. This mapping is accomplished through a *remote-object model* in which an object is defined in terms of a set of interfaces declared in an IDL (Interface Definition Language). Messages to an object can then be redirected to a remote address space. Waldo et al. [65] showed that frameworks that hide the distinction between local and remote objects are inherently unreliable. This is because such a model inevitably leads to an RPC (Remote Procedure Call) type of framework that cannot deal transparently with problems caused by partial failures and concurrency constraints. To assist the application programmer in dealing with these problems, distributed-application development frameworks have been augmented to open up the implementation of remote objects, even though that goes against the principal of abstraction. Various, partially overlapping programming techniques are being used to make the object support environment more open, including reflection [43, 63], (dynamic) composition [59], and AOP (Aspect Oriented Programming) [39, 40]. In essence, both reflection and AOP allow for finer grained separation of concerns, in turn allowing the programmer to pinpoint better where partial failures play a role. Using composition, building larger objects from smaller objects, resembles micro-object programming. Still, all three programming techniques aim to support the transparent promotion of local objects to remote objects by sending interobject messages through a network. Therefor, none solves the

basic design flaw: applications still have to send messages through a distributed environment prone to partial failures. Therefore, all the difficulties inherent to distribution addressed in [65] remain. Other extensions to the object model have been researched. For example, there is the large category of actors. Actors extend the object model by adding concurrency to the distributed object [2]. Basically, actors are autonomous, distributed, concurrently executing objects that exchange messages. Interesting as this may be, it does not solve the problem of partial failures, because actors that are distributed over different nodes in a network will need some form of network communication. Adding concurrency does not solve the problems caused by partial failures. Transparency often is feasible if no errors occur. For example, Parrot is a system that transparently integrates distributed-file services by means of an "interposition agent" [62]. However, ignoring or sublimating errors defies transparency. For example, in the Parrot system, some files (notably FTP (File Transfer Protocol) files) are copied, changed and written back without (transparent) collision resolution. Also, some errors lead to a premature termination of the application process. One could argue that premature termination is transparent to the program, but usually, it is not transparent to the user. Even though total transparency is impossible, it still is possible to help the application programmer with the distributional aspects of programming. A new approach to simplify distributed-systems development using an explicit nontransparent failure model is described in the next section.

3.2 A New Approach

The new approach is based on the assumption that it is better to reverse the flow of information by copying selected remote objects to the local address space for *local* processing instead of sending instructions to remote locations for remote processing. Local processing will prevent the partial failures associated with remote processing. Using this approach leaves information retrieval to be the only thing that depends on remote processing. Moreover, an object retrieval request can either have a positive or negative outcome and false negatives can safely be interpreted as normal negatives. This indifference to partial failure will be referred to as *apparent failure* because it allows the application programmer to know both where and how things can fail. An object change request, on the other hand, does not have apparent failure because it relies on an atomic execute--and--acknowledge step that can fail partially, often promoting false negatives into a big problem for the application. No matter the direction of the information flow, remote objects have to be located. As will be described below,

assigning a systemwide unique ID to remote objects enables a middelware layer to offer an homogeneous *cloud* of remote objects who's exact locations are of no concern of the application layer.

This novel approach to distributed programming is thus based on a copy-before-use scheme, featuring transparent object location and apparent failure. For some application domains, this approach allows massive information exchange by vast numbers of applications without partial failure deteriorating the system. Distribution aspects are handled through a simple API (Application Programming Interface), enabling an application to express its needs irrespectively of what peer applications need. This pure localized expression of distribution needs will result in emergent communication behavior, which turns out to be sufficient to handle many information exchange patterns for which current approaches require explicit control by a programmer. This approach, however, is not without its own challenges, most notably in the realms of concurrency, availability, and performance. A response to these challenges, a replication system featuring a very simple, lightweight object, dubbed *MO (micro-object)*, will be introduced below. The cloud of MOs is provided by a set of distributed *MO cloud servers* or *cloud servers*. The resulting system is called an *MO system*. It's purpose is to provide application programmers with small distributed building blocks (i.e., micro-objects) that can be used to construct larger distributed objects (i.e., application objects).

3.3 Micro-Object Internals

At the heart of the MO system lies the notion of a micro-object. A micro-object is a relatively small container used to ferry copies of distributed data around. For the creation of larger distributed data structures, micro-objects can be clustered into arbitrary graphs. Before delving into the design choices, first a brief, yet complete description of a micro-object, is in order. Figure 3.1 shows its organization.

A micro-object is used to distribute an *immutable* and a *mutable* data part. The immutable part comprises a *token* by which the object can be uniquely identified. Every token contains a *home location* pointing to the home server, where a copy of the micro-object is guaranteed to be kept (until it expires). The token is systemwide unique and can be used to locate a copy of the micro-object. More profound than the immutability of the token, is the fact that it is not the only immutable part: the immutable section also contains a limited-sized buffer of (encrypted) application data, or *payload*. It should be stressed that the payload cannot be modified, a design choice that will be discussed below. Though

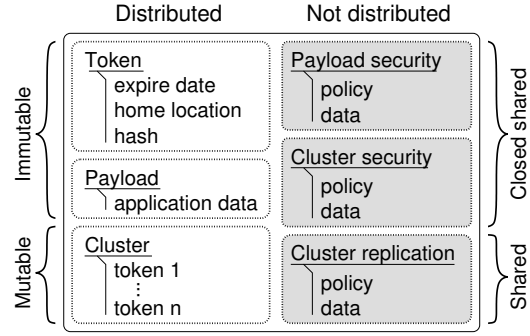

Figure 3.1: The general organization of a micro-object. The distributed part of a micro-object contains information that is shared by all local copies. The part that is not distributed contains local information that can (and usually does) differ per local copy and can only be shared out of band.

the payload (and the token) are immutable, a micro-object can group related micro-objects into a mutable *cluster*. Every micro-object has a cluster of zero or more other micro-objects. There are no constraints, every micro-object can be put in the cluster of any other micro-object. Clustering allow for the construction of arbitrary graphs of micro-objects, which are versatile enough for a whole class of applications. These graphs of micro-objects are limited mutable, they can only grow because a cluster is like an append-only list: members can only be added, but never removed. As discussed below, this restriction simplifies distribution and the development of distributed applications. This kind of limited mutability might seem too limiting, however, in Section 3.5.4, it will be shown not to be the case. The immutable and mutable section together form the *distributed* part of a micro-object. This part is copied and updated across a network by the MO system. An important issue is that the distributed part is securely protected against unauthorized access. Equally important is that the application using a (copy of a) micro-object stays in full control regarding the replication of the mutable part (i.e., the cluster) of that micro-object. An application (programmer) has to express only the policy for replicating the object's cluster according to its own local demands. For example, if rapid dissemination is needed, an application may specify that changes to a cluster (i.e., additions) should be flooded throughout the network. Whether flooding actually takes place depends on the (again local) needs of potential recipients.

This protection and control is achieved through the *nondistributed* part of a micro-object. The nondistributed part consists of two sections. The *closed*

shared section describes how the payload and cluster sections of the micro-object are protected. Typically, this section contains policy descriptors and encryption keys: information that may be disclosed only within a closed group through secure channels. The openly *shared* section describes the rules (i.e., the replication policy) that should be followed when copying (changes in) cluster information to and from other address spaces. By its nature, replication data has to be shareable, however, it does not classify as distributed data, because it does not have to be the same for every individual copy of a given micro-object. As discussed in the next section, these local policies provide a high degree of flexibility in distributing and replicating micro-objects.

3.4 Example Scenario

To illustrate the organization and usage of micro-objects, consider the following simple scenario. Alice, Bob, and Clare regularly publish news messages that they would like to distribute as micro-objects. To this end, Alice takes the initiative to create a micro-object M for storing their shared news messages. The payload of M will not need to hold any data other than perhaps a description of the type of news items it is intended to contain, or maybe a password for the messages associated with M. The cluster of M holds the tokens of micro-objects holding the actual news messages. Note that this makes Alice's dedicated cloud server the home server for M. Alice's local copy of M is denoted as M_A. Alice then passes the token (a systemwide unique identifier) of M to Bob and Clare, using any well known communication method, and Bob and Clare retrieve their copy of M (referred to as M_B and M_C, respectively) from the MO system. Note that the server's contact information (i.e., Alice's dedicated cloud server) is stored as part of the object's token.

Once Bob and Clare have the token of M (and the proper security credentials) they can add a micro-object with a message payload to M. However, by default, these additions to M's cluster will not be forwarded to the other parties. In order to express that additions should be actively forwarded to other parties, Alice, Bob, and Clare decide to choose the replication policy of their local copies of M such that new news items are instantly forwarded to all other participants. Now assume that Bob produces a news item that he wants to share with Alice and Clare. To that end, he creates a micro-object (N_1 at his dedicated cloud server) containing the an actual news message and adds the its token to his local copy M_B of M that will be forwarded to his home server.

Due to the chosen replication policy, the home server will make an attempt to forward any of the elements contained in the object's cluster, as shown in

Figure 3.2(a). The only server it knows about (at this point in time), is the home server of M (i.e., Alice's dedicated cloud server). Bob's server will then contact all the servers in the replication data of M_B, in this case only the home server of M, to report the additions to the cluster of M. This reporting is done by means of an ASSENT request, which essentially initiates a harmonization of cluster elements between M_A and M_B. The word "assent" is used because the harmonization protocol has only one type of message requesting: "please assert to this cluster content." No message, not even an acknowledge message is expected in return. However, if the receiver finds the cluster incomplete, it might send out its own assent request.

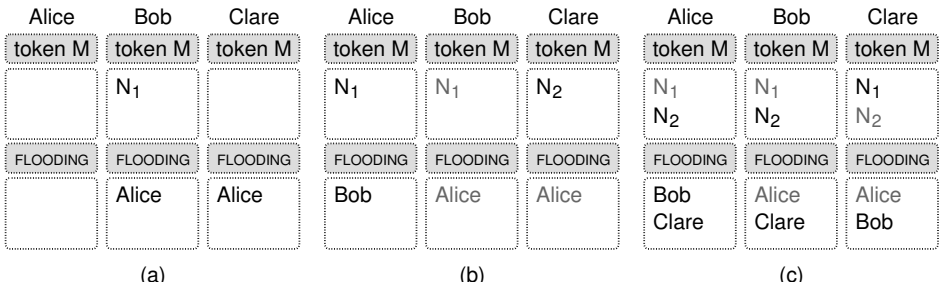

Figure 3.2: Memory snapshot of local copies of M at time (a), (b), and (c). Shown are the fields, token, cluster, replication policy, and replication data.

Only because the replication policy (FLOODING) specified for M_A *matches* the one specified for M_B, will Alice's server react to the ASSENT request. Alice's server will first add Bob's server to the flooding replication data of M_A. Then Alice's server will add the new token for N_1 to the cluster of M_A. After updating M_A, Alice's server searches the replication data of M_A for servers that are not in the replication data of M_B. Since it finds none, the changes need not be flooded onwards to other servers.

Next, assume Clare inserts a news item N_2. At that moment, Clare's server sends out the cluster and replication data of M_C to all servers in the replication data of M_C, as shown in Figure 3.2(b). The only server that is known is the home server of M, Alice's server. Alice's server will then update both the cluster and the replication data of M_A, after which it will search for servers in the replication data of M_A that are not in the recently received replication data of M_C. In this case, there is one candidate: Bob's server. Bob's server is sent the cluster and replication data of M_A, so it can update M_B. Clare's server is also sent the replication data of M_A so it can update the replication data of M_C. The resulting

situation is shown in Figure 3.2(c). From a application programmers point of view, M, N_1, and N_2 make up an distributed application object that is mutable and can be represented by a graph, see Figure 3.3. Note that each node of the

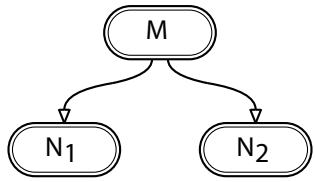

Figure 3.3: Alice's distributed news messaging object represented as a rooted graph. The messaging object contains two messages. Access to the root (i.e., M) is enough to access all the news.

graph is located on a different home server and replicated on all servers and application computers involved.

There are a number of important observations to make. First, note that cluster elements are only tokens. As a consequence, after the clusters of M_A, M_B, and M_C have been merged, the servers of Alice, Bob, and Clare will still need to explicitly fetch N_1 or N_2 to get (the payload of) the new messages.

Also note that news items can be forwarded only to cloud servers that are known to the forwarder *and* that have indicated that they are willing to accept such items by means of a matching replication policy. An important effect of this need for matching is that, for example, Bob cannot produce a news (or any other) item that will be stored at Alice's, Clare's or any other server without cooperation of that server.

It should be stressed that there are only local copies of M in the system. Although M has a home server, this does not mean that there exists a most up-to-date master copy of M (e.g., one of which the cluster has incorporated all changes so far). Instead, it only means that *some* local copy of M is guaranteed to be found at its home server.

Finally, it is worth pointing out that, in the example, the clusters of the local copies of M did converge, as distributed data should. However, the replication data of the local copies of M did not (need to) converge. In effect, only after some elapse of time did all news items reach all interested parties.

In this example, the steps for enforcing cluster security policy were omitted. These steps would (for most cluster security policies) entail authentication to ensure that only those additions to the cluster that are properly authenticated will be accepted. Security policies will be further discussed below.

3.5 Design Issues

The main goal of the MO system is to make it easier for programmers to design and develop distributed applications. Initially, it was conceived as an abstraction layer to develop a unified messaging application. The reasons to make the abstraction layer generic nonetheless, were twofold. First, abstractions work best if they are isolated from others, lest they become leaky. Second, the more generic some thing is implemented the simpler the code. Therefore, unified messaging initiated but does not play a role in the design. The MO system makes it easier to *identify, locate, delete, update, protect,* and *replicate* distributed data by providing a clear and singular way of dealing with these issues. Some of the protection and replication aspects, however, depend on local (temporary) circumstances, and have to be dynamically directed by the application. To this end, the MO system offers a limited number of security and replication policies for the application (programmer) to choose from that can be tuned by changing local security and replication data. The replication data is shared throughout the MO system when necessary, the security data, however, is *closed shared data*, because it is shared in a closed group only. Furthermore, the MO system has no data access control, but is able to detect bogus data to some extent. The design issues concerning all of these points are explained below.

3.5.1 Identifying Data

Tokens are used to manage micro-objects and to describe links between micro-objects. Each micro-object is associated with a token that is systemwide unique to simplify its processing in a highly distributed environment. For example, the token allows tracking an object back to its home location, and also to check how long servers need to support its distribution by means of an *expiration date*. Besides these two attributes, a token also consists of a *hash*, which is computed over the home location, expiration date, and the object's payload. The implementation is slightly more complex than this: to distinguish micro-objects with the same home location, expiration date, and (encrypted) payload, the implementation also features a token counter. Essentially, the hash ensures, with a sufficiently high probability, that the token is indeed systemwide unique. Nonetheless a token can be computed locally: there is no need to communicate with another party. Addressing data directly frees the application (programmer) from having to locate objects. All the objects are just in one big cloud and local copies can be retrieved from just knowing the micro-object's token.

3.5.2 Locating Data

After some data has been requested by the application, the MO system does have to locate it for retrieval. With improperly designed identifiers, data location can become a major issue in any distributed system. The WWW handles massive data exchange and has a simple scheme for locating data: the URL (Uniform Resource Locator). The locating part of the URL is the home location of an object. Basically copying this simple proven concept, each micro-object has a home location (i.e., a server) that is guaranteed to hold (a copy of) the micro-object. Since every micro-object has a home location, the payload of a micro-object can, pending communication errors, always be found, independently of replication efforts. Note that replication improves the QoS (Quality of Service): the micro-object system is fully functional without replication.

Since micro-objects expire, there is little need for the, sometimes relatively intricate, "binding" steps as featured by many existing middleware solutions. However, if need be, the micro-object system can, on a per object basis, use indirect addressing. For example, one micro-object might have a home location containing a DNS host name, another micro-object might have a raw IP address. There is no native binding mechanism but there is no fundamental reason why several indirect addressing, or binding, methods could not be added.

A consequence of this design is that the creator of a micro-object is responsible for keeping it online until its expiration date. One might consider this a drawback, but it introduces a form of fairness as the creators of data should now also provide the resources for keeping their data in the system. In this way, creators hold a bigger share in the cost of resources (CPU time, storage, network bandwidth) in comparison to other approaches, like systems based on NNTP or SMTP.

Still, to make this home location scheme work, the system has to provide the means to retrieve a copy of a given micro-object from its token until it expires. Therefore a home server needs to be always online, just like the WWW depends on servers being online. This scheme has been proven to work, but not to be robust. To enhance the robustness, the MO system contains additional replication options as described in Section 3.5.6.

3.5.3 Deleting Data

Deleting a distributed object means deleting all its local copies. Every micro-object has an expiration date: this helps maximize the number of issues that can be dealt with locally by the MO system. The expiration date specifies until when the object can be retrieved from its home location. The expiration date

should be set to allow a micro-object to reach all participants it needs to reach. The expiration date is determined by the application program(mer). Once the expiration date has passed, the system is no longer required to store the object on any server, most notably its home server. There is now no longer a need for a delete operator. In other words, if a micro-object is ignored long enough, it will disappear. A distributed delete operator could be fairly complex, especially if it would need to guarantee that all replicas of an object had indeed been removed.

In order to keep an object longer than its expiration date, an application will need to explicitly take action, such as requiring its local server to extend the lifetime of the object. It can do so by specifying a local SUSTAIN replication policy. This issue is detailed below.

To prevent premature expiration some form of clock synchronization between all participating parties is needed. The granularity of this synchronization need not be too fine and can easily be satisfied through a time protocol such as NTP. Assuming that the clock of a server can be kept up-to-date with a precision of T time units, a simple solution to premature expiration is to keep every expired micro-object for a grace period T^* time units with $T^* > T$. Note that each server can locally determine its own grace period based on the granularity and precision of its time synchronization mechanism.

3.5.4 Updating Data

In all but the most trivial applications, application objects change, and if the object is distributed, local copies of that object might need to be updated. One of the major challenges of any distributed system is supporting timely propagation of updates of distributed objects. However, it is difficult, and often even impossible for a system to predict which data will be updated, where updates will be needed, and when. This lack of knowledge is unfortunate, as better predictions will enhance the positive effects of replication, such as responsiveness and availability. Since even the application programmer often has a hard time predicting changes, the distributed part consists of a mutable and immutable part, as shown previously in Figure 3.1.

This separation effectively concentrates changes in the mutable part of an object, making them better explicit to both the application (programmer) as well as the MO system. The mutable part (i.e., the cluster) exclusively contains only references to (i.e., tokens of) micro-objects. Allowing only the set of references (i.e., clusters) to change simplifies updates considerably. Even the update operations on the mutable part are limited by design. In particular, there is only an operator to add tokens, but no operator to remove tokens, further

simplifying the update process. Moreover, the mutable part has been specially constructed for efficient replication by sorting its elements on expiration date. This sorting allows us to construct efficient representations of clusters so that two parties can quickly detect differences in their respective clusters [66]. Note that since the expiration date is part of the token, a list of tokens can be sorted locally, in line with the design philosophy to maximize the number of issues that can be decided on locally.

This model forces the application to express distributed application objects as (a growing number of) immutable parts glued together in a way that is efficient for distribution. It can be argued that the combination of an immutable payload and a limited mutable cluster is not enough to allow for distribution of arbitrary mutable application data. However, a broad range of fully mutable distributed application objects can be efficiently supported. A demonstration of this is given in Section 3.7.

3.5.5 Protecting Data

The MO system supports fine-grained security of distributed data, due to the strict separation of security management and object management. Distributing data raises fundamental security challenges. The potential number of people who could access distributed data could be huge and integrity and confidentiality of data are not protected by personal hardware as is possible for nondistributed data. Therefore, additional protection is needed. The MO system uses a combination of end-to-end message encryption and message authentication because it significantly reduces the security demands for remote parties. Both encryption and authentication are needed. Encryption prevents an attacker from knowing the content of a micro-object but does not protect it against manipulation. Authentication can protect against manipulation of data but does not protect against reading data. Authentication is also useful to protect the network against attacks involving data spoofing. It is important to understand that this encryption and authentication scheme is needed to protect the MO system, not the data that is transported through it. The combination of end-to-end encryption and authentication happens to provide some level of protection for application data, but it does not satisfy all security needs of all applications. To name but one example, attacks based on traffic analysis would not be foiled this way. To counter traffic analysis, one might implement a replication policy based on sophisticated cryptographic protocols like mix-networks. This, however, is a whole separate field of research [19] and beyond the scope of this thesis.

3.5.6 Replicating Data

As stated before, the MO system always utilizes a *local copy* of a data object, where the traditional approach is to utilize a *remote copy* of a data object through RPC or RMI. This difference has important implications for data replication. In a traditional system, replication is deployed to enhance performance or availability. As a result, separate mechanisms are needed to support replica placement, consistency enforcement, and redirecting clients to the best replica [58]. Moreover, replication may require the collaboration of third-party servers, leading to the incentives and fairness problems hampering many of today's decentralized peer-to-peer systems [44, 54].

Basic Replication

In a local-copy system such as the MO system, purposefully replicating objects for availability and performance can come at virtually no extra costs. First, in order to access an object, an application will have to make a local copy of that object. This mandatory object pulling on-demand is the *basic replication* method of the MO system. As a result of basic replication, micro-objects are already massively replicated on-demand, by client applications.

Additional Replication

In addition, if an application strives for higher performance, robustness, or availability, it can specify this by means of an additional object-push or object-prefetch replication policy, to be executed by the dedicated cloud server. The effect of such *additional replication*, is that objects end up quicker at those cloud servers where they are wanted most. Effectively, additional replication allows cloud servers to preemptively fetch or push objects anticipating their use by local applications. Note that additional replication is established as an ad hoc agreement within a group of collaborating local applications using matching replication policies, whereas basic replication is supported by all cloud servers, independently of applications. One could view the MO system as a generic structure that facilitates the creation of ad hoc overlay networks.

Because of the multitude of additional replication policies, there is no need for a strong one-size-fits-most basic replication. So having additional replication allows relaxation of demands put on the basic replication. Also, basic replication relaxes the demands on additional replication. For example, assume a group of applications jointly follow a gossip-based dissemination and replication of their objects by applying an anti-entropy protocol [27]. These protocols are known to disseminate data in a robust way, but may easily introduce inconsistencies as

different nodes will see different values. Since the MO system can always rely on basic replication, these problems are alleviated when gossiping is used as an additional way to replicate objects. If the payload of a micro-object is needed immediately, it can always be fetched from its home server, because of the basic replication support of the MO system.

At first it might seem odd to allow a subset of cloud servers to engage in an additional replication policy. In fact, it is one of the strongest points of the MO system. For example, imagine a distributed file system based on the MO system and assume several applications need to access the same file at the same time. In this case, it would make perfect sense to let those applications use an additional high-cost, high-performance replication policy on the micro-object tree that makes up that file. The home servers of these applications would effectively form an additional overlay network.

Eventual Consistency

Micro-objects themselves are immutable. Any changes in an MO's state stems from additions to its cluster. Given that all tokens in the cluster have a systemwide unique token, it follows that it is possible to consistently sort any given set of tokens the same way. By sorting the tokens of a cluster upon insertion, the order of token insertion becomes irrelevant. From a consistency point of view, this is a very nice feature, because it greatly simplifies keeping different local copies of a cluster consistent. Any replication strategy that delivers all the cluster tokens to all the local copies at least once will eventually cause all local copies to be consistent, even if several clusters are delivered out of order and multiple times. Analogously to the definition given by Tajibnapis [60], clusters in the MO system can be said to enjoy eventual consistency.

Heterogeneous Replication

It is worth noting that the basic replication takes place all the time. Therefore, it takes place in parallel with any additional replication. In most systems using replication, having two replication strategies run in parallel could cause sophisticated challenges. At the very least the replication policies have to be aware of the fact that other replication could be taking place. A lot of replication policies are based on the silent assumption that there are no out-of-band or "sudden" changes. The MO system is designed to allow such parallel heterogeneous replication. This is relatively simple, because payloads are immutable and clusters have no delete functionality. Not only can an additional replication policy be combined with the basic replication, but a micro-object can have several het-

erogeneous replications going on in parallel. Note that it makes sense to allow this heterogeneous replication because without it, applications would have to be aware of what other applications are doing if the data they use overlap. This would complicate application development enormously while the whole point of developing an MO system is simplification of application development. By design choice, one local copy can have only one additional replication policy. This makes the API more intuitive. However, by making multiple local copies of a micro-object, a micro-object can have heterogeneous replication policies running at the same time.

3.6 Systems Design

Now that the micro-object has been introduced, it is time to discuss the design of the MO system. All the components of the current version of the micro-object system are available for download from micro-objects.org. The infrastructure of the MO system is not unlike the E-mail system in the sense that a distributed application does not directly contact other applications. Instead, a network of (dedicated) servers is used for distributing micro-objects.

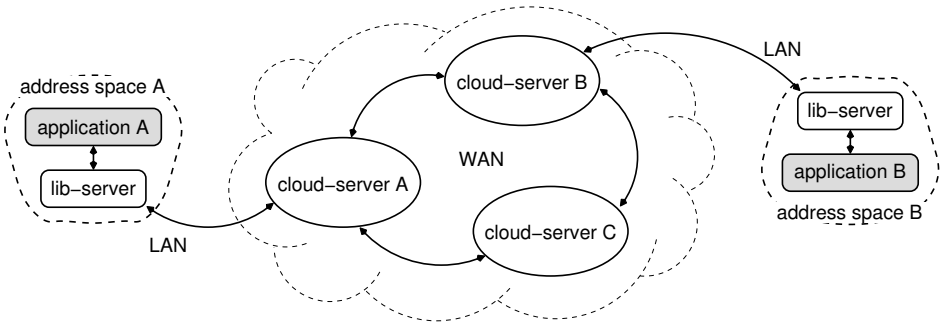

Figure 3.4: Overall design of MO system.

An application contacts a local server, much like an E-mail client application would up- and download new messages to and from a server provided by a company or ISP. These servers will communicate as peers to distribute micro-objects. Just like the E-mail system, an application can be offline without disrupting any ongoing replication scheme.

Unlike the E-mail system, however, the MO system does not do end-point delivery. In delivering information, it is more like the traditional WWW model: information is stored in a single known place and can be cached closer to the

destination. Like WWW proxy caching, multiple applications will generally be using the same server cache for a better cache hit ratio.

On top of this basic "pull on demand" replication, the MO system features additional dynamic replication policies. The application (programmer) can specify when a server needs to spend additional resources on replicating a specific micro-object.

To handle basic and additional replication, the MO system follows the classical three-tier approach. The three tiers consist of the *application*, the *lib-server*, and the *cloud server* (as shown in Figure 3.4). The first tier, the application, shares its address space with the second tier, the library server, also referred to as the lib-server. The second tier, the lib-server, provides library functions and spawns process threads acting like a server, hence the name. The lib-server communicates with the third tier, its dedicated cloud server, through a relatively secure and fast connection, for example, a LAN. The cloud server has to be always online whereas the lib-server can be regularly offline. The next section contains a closer look at the cloud server and the lib-server.

3.6.1 The cloud server

The cloud server, sketched in Figure 3.5, fulfills three major roles. First, the cloud server has to *store* every micro-object that a trusted MO application has created. The server will store such an object until it expires, thus acting as the object's home server. Second, it has to *cache* incoming micro-objects. Third, it has to run threads to *execute replication policies*. The store and cache differ

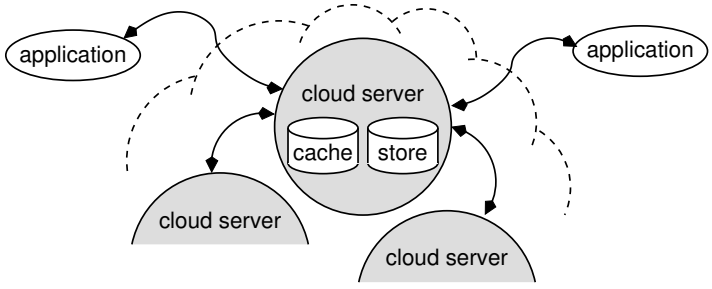

Figure 3.5: Close-up of the cloud server.

mainly only in how they clean up their contents. Micro-objects can be removed from the store only if they have expired, while they can be removed from the cache at any time. Just as with additional replication, cache management has no external dependencies as a discarded micro-object can always be retrieved

from its home server. A cloud server has a remote and a local communication channel. The local channel differs from the remote channel in the sense that it can be made as trustworthy as needed, for example, by means of SSL/TLS. In practice though, the local channel will simply be a LAN or ISP network offering low latency and possibly also high bandwidth. The cloud server is a basic request/response system.

A *remote request/response* sequence is used by the cloud servers to communicate with their remote peers. There are several types of such communications. For example, if one cloud server needs a micro-object, it can ask any other cloud server for it by sending the latter a FETCH request containing a valid token. If the receiving cloud server has the requested micro-object (in its cache or store), it can send the micro-object back in response. Since encryption is used on a higher level, there is no need for security checks, most notably there is no distributed infrastructure for security. It will also be difficult to forge a valid token, because the token space is very sparsely filled.

To facilitate load balancing and additional replication policies, a remote response can contain further information by means of a *ditto-list*. A ditto-list contains a number of cloud servers that are likely to have the requested information. In general, a cloud server is put on a ditto-list because it has previously made a similar request. So, it is very likely that any server on the ditto-list will be able to respond to the request. Not only is the original request sent to the cloud server in the ditto-list, but also the related, subsequent requests. This will save the original home server a whole set of requests.

Any remote request can trigger a BUSY response with a ditto-list. This reply indicates that the cloud server is swamped with requests. It is then up to the requesting cloud server to re-route the remote request to another cloud server from the ditto-list. Note that this solution is not new, sometimes it is applied to alleviate hot-spot problems in the Web (see, e.g., [50]).

The ASSENT request is sent whenever two cloud servers want to make their respective copies of a micro-object consistent, that is to make sure that the two clusters are the same. To this end, a cloud server A can send a ASSENT request to cloud server B containing micro-object M. This request will allow B to possibly merge the elements contained in A's copy of M's cluster with its own copy of M's cluster. B can now also detect which elements are missing from A's copy of M's cluster and pass this information back to A. If both A and B decide to add the missing elements to their respective copies, the two will be the same after the ASSENT exchange. After merging, A or B might decide to forward information to other servers, as seen in the example in Section 3.4. Note, however, that each party is completely free to decide which elements to include in its local copy of

M.

A *local request/response* sequence is used by the cloud server to communicate with a (trusted) lib-server. An obvious local request is the FIND request, which is issued by the lib-server. It forces the cloud server to find a requested micro-object, either in its cache or store or by means of a remote FETCH request. If the cloud server is the home for the requested micro-object (and it has not expired), it will find the micro-object in its store. If not, the cloud server can use a remote FETCH request to a peer cloud server, most notably the home cloud server of the requested micro-object. It will forward the response to the lib-server, but also extract the micro-object from the response and put it in its cache. Note that in this case the cloud server acts like a proxy server. As with the FETCH request, there is no need to check for access permissions.

Only a trusted application can ask a cloud server to become the home of a micro-object. It does so by sending a local ADOPT request. If local policies allow it, the cloud server will put the micro-object in its store.

Also, only a trusted application can ask a cloud server to start (or stop) executing a replication policy for a given micro-object. It does so by sending a local REPLICATE request to the server. If local policies allow it, the cloud server will start the requested replication policy for the given micro-object, pushing its payload and cluster to other cloud servers, the same goes for new additions to the micro-object's cluster while the replication policy is active. Micro-objects in a cluster inherit the replication strategies of the micro-object holding the cluster, so their payloads and clusters are replicated as well. This recursion stops at an application specified level. Replication example code will be given in Section 3.6.4. Note that, as stated in Section 3.5.6, several replication policies can be active at the same time for a given micro-object. Therefore the cloud server has to be able to handle replication data of multiple replication policies per micro-object.

Trusted applications are also allowed to send a local UPDATE request. Such a request contains one or more tokens that are to be added to a given object's cluster. If the micro-object in question is in the cache or store, its cluster is updated immediately. Also if there are replication policies active for this micro-object, they are run, because the arrival of new cluster members may necessitate some action.

The *store* of a cloud server holds all the micro-objects that are at home at that server. However, the store can be populated with "foreign" micro-objects too. To understand why, note that every replication thread has full (i.e., both read and write) access to the store. Consequently, a replication policy like SUSTAIN, by which an object is stored beyond its expiration date could put such a foreign

micro-object in the store. The result would be that this foreign micro-object will not be removed from the MO system until its extended expiration date. Future replication strategies might have other reasons to put micro-objects in the store, for example, to save them from cache cleanups. Note that every cloud server can have its own policies for storage, most notably it could feature a quota system, disallowing or charging excessive usage.

Since every cloud server has to act like a proxy server because all micro-objects are retrieved indirectly through a cloud server, all cloud servers feature a micro-object *cache*. Appropriate caching algorithms for cloud servers still need to be investigated in detail. For now, an LRU (Least Recently Used) algorithm is assumed. Note that the cache algorithm is a local affair, every cloud server can make its own local decisions. For example, it could decide to cache requested micro-objects depending on which application issued the request.

As mentioned the SUSTAIN replication policy is special because it postpones the expiration of a micro-object past its expiration date. Basically, an application (programmer) can ask a cloud server to sustain a local copy of a micro-object for a limited time (but not forever). Note that an application needs to sustain the micro-object at regular intervals. If a micro-object is sustained on its home cloud server, it will still be available to all other cloud servers. If, however, a micro-object is sustained on a set of cloud servers not including the home cloud server, servers outside that set will not be able to fetch it anymore. A prime candidate for prolonged sustaining, for example, would be the root of a distributed file system. Note that this does not imply that an application has to be always online, but only frequently enough to prolong an object's lifetime.

3.6.2 The lib-server

The lib-server is linked into the application's address space as a library. It provides the API of the MO system. Besides a library with functions, however, it also runs separate threads (in the background) in the application's address space, acting like a server. By putting the lib-server in the same address space as the application, it has the same trust level. This makes it simpler for the lib-server to safely access security information like encryption keys.

The API of the MO system (as implemented by the lib-server) consists of one ADT (Abstract Data Type) per concept, each with their own prefix, offering functions like `alloc()`, `free()`, `copy()`, `cmp()`, `put()`, and `get()`. Figure 3.6 lists the ADTs (with their prefix). Rather than object classes, ADTs are used to define the API. This is simply because the implementation is done in standard C for maximum compatibility and platform independence. There is no reason

memory buffer	(bfer_)
home location	(hloc_)
expiration date	(xpir_)
token	(tken_)
payload	(plod_)
cluster	(cter_)
payload security	(psec_)
cluster security	(csec_)
replication	(repl_)
micro-object	(mo_)

(a)

Figure 3.6: Lib-server ADT list.

why micro-objects could not be implemented as computer language objects in an object oriented programming language. The lib-server is based primarily on the micro-object ADT (with prefix mo_). A number of pivotal functions, as shown in Figure 3.7, will be described below in more detail. Throughout

```
mo_create_new(mo_t*, xpir_t, plod_t, psec_t, csec_t);
mo_get_tken(tken_t*, mo_t);
mo_create_copy(mo_t*, tken_t, psec_t, csec_t);
mo_put_repl(mo_t*, repl_t);
mo_get_cter(cter_t*, mo_t);
mo_put_cter_clbk(mo_t*, cter_t*, clbk_t, void*);
mo_cter_add_tken(mo_t*, tken_t);
```

Figure 3.7: List micro-object API functions (partial).

the rest of this chapter, several C-code examples are given. Note that this code blatantly ignores the return value of API calls. For example, the code:

```
plod_put_string(&plod, "Hello World!");
```

does not take into account that this API call could fail. A better, but maybe too draconian way to check the return value would be:

```
assert(plod_put_string(&plod, "Hello World!"));
```

For clarity, these assertions or other forms of return value checks, have been omitted in the example code given in the rest of this chapter.

Mo_create_new()

The mo_create_new() API function creates a new micro-object. For example:

```
/* Set expiration date, payload contents, and security policies.
*/
xpir_put_secs_to_live(&xpir, 120);
plod_put_string(&plod, "Hello World!");
psec_put(&psec, psec_poln_asym, NULL);
psec_put(&csec, psec_poln_sym, NULL);

/* Make some room for the new micro-object.
*/
mo_alloc(&mymo);

/* Create a new object.
*/
mo_create_new(&mymo, xpir, plod, psec, csec);
```

When `mo_create_new()` is called, the lib-server fills the freshly allocated local micro-object with copies of the given payload and security policies. It generates a systemwide unique ID, that is a token, using only locally available information like: the address of the home server, the given expiration date and a hash of the immutable fields of the micro-object. The lib-server will also put the finished local copy of the new micro-object on the outbound queue for the home server. After `mo_alloc()` and `mo_create_new()` have returned successfully, a new local copy has been created from the pool of available memory within the application and lib-server's address space. Also a copy of the new micro-object has been put on the queue to the cloud server, for persistent storage on the home server. This happens independently of any replication policy, since the new micro-object cannot belong, yet, to any cluster, nor does it have a replication policy of its own. This is the only time a payload or security policy can be set. After calling `mo_create_new()`, the given payload and security policies, as well as the calculated token, are bound to this one micro-object, they have become immutable, because by definition, they are part of the immutable part of a micro-object, as shown in Figure 3.1.

Mo_get_tken ( )

With the function `mo_get_tken()` the token can be retrieved from a given micro-object. Tokens can be exchanged out-of-band, but they have to be marshalled first. For example, consider this code:

```
/* Allocate and fill new token.
*/
tken_alloc(&tken);
mo_get_tken(&tken, mymo);

/* Convert the token to a Radix64 encoded buffer for exchange.
*/
tken_to_bfer(&bfer, tken);
bfer_encode_radix64(&bfer, bfer);
```

After the above call to `mo_get_tken()`, the newly allocated token is copied from the token from the micro-object `mymo`. After getting the (machine dependent) token, it has to be marshalled and coded so it can be shared out-of-band. After the `tken_alloc()` and `mo_get_tken()` calls return successfully, a new token has been created from the pool of free memory in the address space of the application and the lib-server. This new token is an exact copy of the token from the micro-object `mymo`. The cloud server does not need to be notified, so no communication takes place between the lib-server and the cloud server as a result of these calls. Once the encoded token is received by another application, it can be used to build the entire micro-object. To rebuild the entire micro-object, or more correctly put, to create a complete local copy of the same micro-object, the token has to be decoded and combined with the correct security policies. The next API function (`mo_create_copy()`) is used for the actual construction.

Mo_create_copy ( )

The function `mo_create_copy()` is used to construct a new local copy of an existing micro-object from its token and security policies. For example, if a token has been received, and the security policies are known, a local copy of a micro-object can be retrieved using the following code:

```
/* Convert the Radix64 encoded buffer to a new token.
*/
tken_alloc(&tken);
bfer_decode_radix64(&bfer, bfer);
tken_from_bfer(&tken, bfer);

/* Create a new local copy using the common security policies.
*/
mo_create_copy(&mymo, tken, psec, csec);
```

First, memory is allocated for a new token. Then the buffer is decoded and demarshalled into the new token. Note that these steps are the reverse from the

steps taken in the example for `mo_get_tken()`, above. With the new token, and some (apparently) known security policies (e.g., a decryption key), a new local copy of the micro-object is created. This object can be subsequently used to get the payload, or add tokens to. After the successful return of `mo_create_copy()`, the application has a local copy of the complete micro-object. The micro-object could have been known to the lib-server before, in that case, no communication has to take place before the call can return. If the lib-server does not have the requested micro-object, it asks a nearby cloud server to provide one, and it holds the application process (or more precisely the requesting application thread), until the cloud server returns an answer. In some cases the cloud server might have a copy of the requested micro-object in its cache. If not, the cloud server will contact other cloud servers, for example the requested micro-object's home server. The micro-object might be stored in the micro-object cache of the cloud server for quicker retrieval in the future.

Note that the security policies used to create a new local copy do not need to be exactly the same as the policies that were used to create the micro-object. In fact, the security will often be a "cut down" version allowing only reading. For example, the `psec_t` argument needs to contain only the decode key in case an asymmetric encryption policy was used. To illustrate, here is some code that "cuts down" the capabilities of a policy:

```
/* Create a new policy for a number of related micro-objects.
*/
psec_alloc(&psec_full);
psec_put(&psec_full, psec_poln_asym, NULL);

/* Alloc and fill a read-only payload security.
*/
psec_alloc(&psec_ro);
psec_copy(&psec_ro, psec_full);
psec_command_("del post key", &psec_ro, NULL);
```

First, the code above generates a new payload security policy named `psec_full`. It is asymmetrical, hence it has a read- and a post-key. Note that the application does not need to generate a key pair first. All initialization is done by the one call to `psec_put()`. This is in line with the simplicity goal. Calling the function `psec_put()` will generate a new encryption key pair, along with any necessary meta data. After creating a new payload security, the above example code generates a payload security policy called `psec_ro`, that would allow another application to read the payload of a micro-object that uses the `psec_full`. However, the MO system would not allow the usage of `psec_ro` to be used to create a new micro-object. In fact, it would be nearly impossible to do, even

if the MO system would want to do that, due to the underlying asymmetric encryption.

Mo_put_repl()

The `mo_put_repl()` function will change the replication policy of the cluster of a local micro-object. Changing the replication policy of a local micro-object is relatively simple with code like this:

```
/* Allocate and set a replication policy.
*/
repl_alloc(&repl);
repl_put(&repl, cter_poln_cyclon, 3);

/* Change replication of the micro-object.
*/
mo_put_repl(&S, repl);
```

The above example code, allocates and sets a replication policy `repl`, to cluster policy number (`cter_poln_cyclon`). The policy number is used by the lib-server to select a block of replication code for this local copy of `S`. This will make the local micro-object `S` start participating in replicating its cluster members with an epidemic protocol named CYCLON, which basically is an enhanced gossiping protocol [64]. After the call `mo_put_repl()` has returned successfully, the given replication policy will have been put on the output queue to the cloud server. The cloud server does the lion's share of the actual replication. This also implies that the replication is not dependent on the application and the lib-server. In other words, if the application quits, the replication can continue. Note, however, that it is thinkable that some stringent cluster security policy might confine (part of) the replication process to the (trusted) lib-server. In that case, stopping the lib-server will cause the replication to be ceased or hampered. Such a strict cluster security policy has not been defined or implemented yet, but there is no reason why it could not be.

Replication Level

Besides the type of replication (in the above example: `cter_poln_cyclon`), the API call `repl_put()` also contains a replication level. In the code above, it is set to `3`. This replication level plays a crucial role in the efficiency of the replication. As noted before, each micro-object has a cluster of tokens of other micro-objects. From an application point of view, these clusters are used to form arbitrary graphs of micro-objects to represent distributed application objects. Given the hierarchical nature of most application structures, the tree depicted

in Figure 3.8 could be a typical example of some distributed application object. Some micro-object S has one child A, that in its turn has B and C as children,

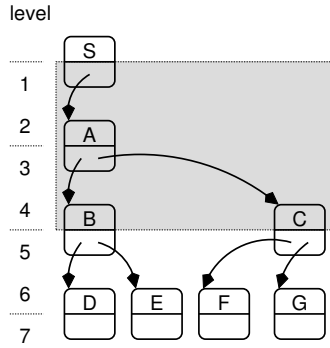

Figure 3.8: Example hierarchy of some micro-objects that form an application object.

etc. Had the replication-level argument in the `repl_put()` call been zero in the example above, only the cluster of S would have been replicated using the replication policy. Had it been `1` then the cloud server would also start retrieving and caching the actual micro-objects corresponding to the tokens in the cluster of S. In the example of Figure 3.8 that would be A. Had it been `2` then the cluster of micro-object A would also be replicated by the cloud server. In casu the tokens for micro-objects B and C. With the given replication level of `3`, the cloud server will retrieve the full micro-objects S, A, B, and C, plus any other micro-object that is added to the cluster of S and A, but not B or C. In effect, the whole tree underneath `S` is replicated up to three levels deep. Replication here means that data is pushed and/or pulled to the likings of the selected replication code. With a tree-shaped graph, this replication level provides an intuitive and generic relation description that allows for more efficient replication. The increase in efficiency of the replication is possible because the cloud server has more (meta) information on what micro-objects the application presumes, are prone to be used in conjunction. If the graph is not a tree but some arbitrary graph containing cycles, the cloud server will not loop indefinitely because any loop can be detected and cut applying one simple rule: any micro-object to be replicated according to a given policy and level, can be ignored if it is already being replicated with that same policy and any level equal or higher to the given level. Changing the replication policy of one local copy of a micro-object, could force the lib-server and the cloud server to start prefetching cluster additions. This is, of course, the very point of added replication. Changing the replica-

tion policy of several local copies, spread around the MO system, would force a subset of cloud servers to start communicating amongst each other. This subset of servers doing extra communication could be said to form an overlay replication network. The emergent behavior of such an overlay replication network is dependent on the details of the replication policy and its implementation. In general it will lead to a higher diffusion of the payloads and clusters of the micro-objects involved.

Mo_get_cter ( )

With the `mo_get_cter()` API function, an application can get a snapshot of the current cluster of a micro-object. The usage is straightforward as can be seen in the first two API calls in the next example:

```
/* Allocate room for a new cluster and get a copy.
*/
cter_alloc(&my_cter);
mo_get_cter(&my_cter, the_mo);

/* Do something with all the tokens in the cluster.
*/
tken_alloc(&tken);
cter_enum_create(&enumb, my_cter);
while(cter_enum_get_next_tken(&enumb, &tken, NULL))
  do_some_thing(tken);
```

The example code above allocates room for a new cluster called `my_cter` and calls `mo_get_cter()` to get a snapshot copy of the current cluster of `the_mo`. Note that a snapshot could already be outdated upon successful return of `mo_get_cter()`. If the cluster of `the_mo` gets updated after the snapshot `my_cter` has been returned, there will be a difference in the cluster of `the_mo` and the snapshot `my_cter`.

The subsequent four lines of code demonstrate a usage of the cluster's snapshot. Using a new enumerator named `enumb` the API call `cter_enum_get_next_tken()` is repeatedly called to iterate through all the cluster members of `my_cter`. Presumably the function call `do_some_thing()` does something with a token. For example, it could get a local copy of the corresponding micro-object's payload using `mo_create_copy()` followed by `mo_get_plod()`. The lib-server will allocate and fill the cluster with a copy of the current micro-object's cluster. No communication with the cloud server is needed. Since the lib-server just gives out (a snapshot of) what it has at the moment of calling, there can be no unexpected delays or other problems caused by partial failures.

Additions to the cluster of the micro-object will not be "magically" added to the cluster my_cter. This makes using clusters of micro-objects simple. There will be cases where the application would want to do more than take a snapshot and iterate the cluster members. For example, it might want to do something with newly arrived cluster members as soon as possible. For this, a somewhat more complex API function exists: mo_cter_clbk().

Mo_put_cter_clbk()

The API call mo_put_cter_clbk() is used to ask the lib-server to call a given function every time the lib-server becomes aware of new additions to the cluster of a given micro-object. A simple callback function could look like this:

```
/* This a very simple callback function.
*/
void my_clbk(mo_t mo, tken_t tken, void *user)
{
  printf("%s.\n", (char*)user);
}
```

A callback function my_clbk() is defined. It returns no value (i.e., its type is void) and accepts a micro-object, a token, and a void pointer as arguments. The callback function is used as an argument to the setup function mo_put_cter_clbk(). For example, like this:

```
/* Start calling my_clbk() for every addition to the cluster of my_mo.
*/
user_data = "Seen new token";
mo_put_cter_clbk(&my_mo, &cter_track, my_clbk, user_data);

do_other_stuff();
sleep(10);

/* Stop calling the callback function for this micro-object.
*/
mo_put_cter_clbk(&my_mo, NULL, NULL, NULL);
```

Once the setup is completed, every new token that is added to the cluster of my_mo will trigger execution of the callback function my_clbk(). The lib-server, or more precisely one of the server threads of the lib-server, will execute the callback function supplying it with the micro-object my_mo, the new token whose arrival triggered the callback, and the user_data argument copied verbatim from the setup routine mo_put_cter_clbk(). Callback functions will be executed by lib-server threads while the main thread of the application does other stuff or sleeps. The tracking of the cluster of the micro-object my_mo is stopped, by changing

its callback function to NULL. Like any local copy of a micro-object, my_mo can have only one callback function active at the same time. However, every local copy of my_mo can, simultaneously, activate its own separate callback function. Modifying the above code to simultaneously activate two callback functions on my_mo, using mo_create_copy(), is left as an exercise to the reader. The callback function will be called by the lib-server, and no additional communication with the cloud server will be necessary. Note that the lib-server has to run at least one thread for this to work. In fact, the lib-server runs several threads and can start and stop additional threads if circumstances dictate it. The code below shows a slightly more complex callback function.

```
/* Print a dot every tenth new token.
*/
void my_clbk(mo_t mo, tken_t tken, void *user)
{
  int *p_cnt = (int *)user;
  if (0 == p_cnt[0]++ % 10)
    printf(".");
}
```

It increments the user data, interpreted as (a pointer to) a number. If this counter reaches a tenfold a '.' is printed. This callback function clearly is not thread safe. Since this might be the case for most nontrivial callback functions, every callback is protected by a lock. There is one caveat, the lock is bound to a single local copy of a micro-object. So, if one callback function were to be shared by several local micro-objects, additional locking of shared resources might be necessary.

The API call mo_put_cter_clbk() takes, besides the micro-object pointer (&my_mo), callback function (my_clbk), and user data pointer (user_data), a fourth argument, a cluster called cter_track. If non NULL, this cluster keeps track of the tokens that should not trigger a callback. Typically, these tokens have triggered a callback already. The lib-server will update the tracker as it makes callbacks. Upon calling the mo_put_cter_clbk() function to start the callback process, the tracker cluster contains all the tokens that are not to be considered new. Upon stopping the callback process, it contains the same tokens, plus the tokens that have triggered a callback. This way an application has more control over the callback process. The code below demonstrates the usage of such a tracker cluster.

```
int cnt = 0;

/* Allocate room for a new cluster and get a copy.
*/
cter_alloc(&tracker);
mo_get_cter(&tracker, the_mo);

/* Do something with all the tokens in the cluster.
*/
tken_alloc(&tken);                                  /* Redundant. */
cter_enum_create(&enum, tracker);                   /* Redundant. */
while(cter_enum_get_next_tken(&enum, &tken, NULL))  /* Redundant. */
  do_some_thing(the_mo, tken, &cnt);                /* Redundant. */

/* Start calling my_clbk() for every addition to the cluster of the_mo.
*/
mo_put_cter_clbk(&the_mo, &tracker, do_some_thing, &cnt);
```

A `tracker` cluster is allocated and filled with a snapshot of the current cluster of `the_mo`. For each token in the tracker cluster, the function `do_some_thing()` is called. After that a callback is requested on the cluster of `the_mo`. Without explicitly tracking which tokens have been processed and which not, all kind of race conditions could arise, leading to repetitive callbacks on a single new token or callbacks that should be preformed but are not. The solution is straightforward, there will be no callback for the tokens that are in the `tracker` cluster. Only tokens that have arrived (and will arrive) after the tracker was filled, will cause the callback function `do_some_thing()` to be called. Note that had tokens arrived in the time slot between the call to `mo_get_cter()` and the call to `mo_put_cter_clbk()` they will be stored in the cluster of the micro-object, but not in the `tracker` cluster. Hence, the callback function will be called on each of those newly arrived tokens as soon as `mo_put_cter_clbk()` is called. In fact, using an empty tracker will cause execution of `do_some_thing()` on all (current) cluster members of `the_mo`. In the example code above, the block of code marked with `/* Redundant. */` could simply be removed, without effecting any change in behavior of the code. This is because if `mo_put_cter_clbk()` is called with an empty tracker, the callback function `do_some_thing()` will be called on all the current members not in the tracker.

In an extreme case a single application might even have many local copies of one and the same micro-object, each with a different callback function. A single addition to the cluster of such a micro-object would then trigger several callback functions. The current implementation does not guarantee the order in

which the callbacks are executed, nor does it limit the number of simultaneous callback executions. The code below could be added to the example code above:

```
/* Add a callback function doing other stuff with the cluster members.
*/
cter_alloc(&trck2);
mo_alloc(&copy_mo);
mo_copy(&copy_mo, the_mo);
mo_put_cter_clbk(&copy_mo, &trck2, do_other_stuff, &cnt);
```

This would cause the callback function `do_other_stuff()` to be called for each token in the cluster of `copy_mo` and thus also of `the_mo`. Upon arrival of a new cluster member, both `do_some_thing()` and `do_other_stuff()` would be called. Note the usage of the "standard" `mo_copy()` to create a new local copy of `the_mo`, without the replication and callback state.

Mo_cter_add_tken ( )

The examples so far showed only how the MO system reacts to newly added tokens. Actually adding tokens is done using the `mo_cter_add_tken()` API function. The example code below shows a way to add `N` micro-objects to the cluster of a micro-object.

```
/* Add the tokens of N new micro-objects to the cluster of the_mo.
*/
for (i = 0; i < N; i++) {
  mo_create_new(&tmp_mo, xpir, plod, psec, csec);
  mo_get_tken(&tmp_tken, tmp_mo);
  mo_cter_add_tken(&the_mo, tmp_tken);
}
```

The snippet of example code above repeats three steps `N` times. Presumably an expiration date (`xpir`), payload (`plod`), and two security policies (`psec` and `csec`) have been allocated and set so they can be used to create a new micro-object `tmp_mo`. From this new micro-object its token is extracted using `mo_get_tken()` and subsequently added to the cluster of `the_mo` using the API call `mo_cter_add_tken()`. The last two calls in the loop extract a token and add that to a cluster.

Extracting a token to add it to a cluster is such a common combination that there exists an API call `mo_cter_add_mo()` that does the same thing in one go. Note that adding this API call sins against the goal of keeping the API interface small. However, because it hardly adds any complexity, it was incorporated. This example does not do something really useful, because all the micro-objects

added carry the same payload and expiration date. Note, however, that they will be distinguishable because they each have a systemwide unique token.

One would usually expect the payload to differ for the individual cluster members. In practice one would also expect that the order of the cluster members could be important. Since the expiration date is the most significant part in the sorting order, incrementing it before each call to `mo_create_new()` will guarantee the sorting order. For this a special API function called `xpir_uinc()` exists. This function will increment the expiration date by one microsecond. The code below demonstrates the usage of `xpir_uinc()` and `mo_cter_add_mo()`:

```
/* Add tokens to the cluster of the_mo in a given order.
*/
for (i = 0; i < N; i++) {
  xpir_uinc(&xpir);
  mo_create_new(&tmp_mo, xpir, plod, psec, csec);
  mo_cter_add_mo(&mo, tmp_mo);
}
```

Adding a token to the cluster of a micro-object causes the lib-server to not only update its own data structure, but also to send an update request to the cloud server. The cloud server will also update its internal state and, dependent on active replication policies, will forward the update to other cloud servers and lib-servers.

Actually, as discussed in the above section on `mo_put_repl()`, whether or when a local cluster update triggers a remote cluster update is dictated by the active replication policies on the cloud servers involved.

3.6.3 "Hello World!" With Micro-Objects

To illustrate programming with micro-objects, let us consider the standard "Hello World!" example. Note that the code given in Figure 3.9 up to and including Figure 3.12, are excerpts from a demonstration program as given in appendix A.

The function `set_msg()`, shown in Figure 3.9, creates a systemwide visible micro-object, that expires in 24 hours. Note how an application first prepares the content of the object: it creates an application-level micro-object and passes payload and security data to the lib-server. The actual creation of the system-level micro-object takes place by a call to `mo_create_new()`, effectively placing the object in the store of its home server. After this creation, it becomes possible to refer to the object by means of its token, which is generated by the home server. Note that at this point that application-level object can be destroyed: the intended micro-object is now successfully being taken care of by the underlying MO system.

```
void set_msg(bfer_t *p_bfer)
{
  mo_t   mo;
    :
    :  /* Code omitted for clarity. */
    :  /* Full code in Appendix A. */
    :
  /* Allocate ADTs. */
  mo_alloc(&mo);
    :
    :
  /* Fill the components for the micro-object. */
  plod_put_string(&plod, "Hello world!");
  xpir_put_days_to_live(&xpir, 1);
    :
    :
  /* Create the micro-object. */
  mo_create_new(&mo, xpir, plod, psec, csec);

  /* Fill the output buffer with the radix64 of the token. */
  mo_get_tken(&tken, mo);
  tken_to_radix64_bfer(p_bfer, tken);

  /* Deallocate ADTs. */
  xpir_free(&xpir);
}
```

Figure 3.9: Creating a "Hello World!" micro-object.

Let us now consider how another (or the same) application can obtain this object and retrieve its "Hello World!" message.

The function `get_msg()`, shown in Figure 3.10, retrieves a local copy of the "Hello World!" micro-object and copies its payload. Micro-objects are distributed, so that `set_msg()` can be run in one address space, while `get_msg()` can be executed in another (or the same) address space. For this to work, the payload security of both processes needs to match. When using asymmetrical encryption, for example, the function `set_msg()` needs to provide at least the encryption key (also referred to as *post-key* in [67]) in `psec`, while the function `get_msg()` needs to provide at least the decryption key (also called *read-key*) in its `psec`. These keys have to be exchanged out-of-band, for example, through a secure channel.

Figure 3.10 again illustrates locality when using micro-objects. An application starts with constructing an application-level object, providing the required token and necessary security information. The actual object retrieval takes place by means of the call `mo_create_copy()`, which instructs the receiver's dedicated cloud server to fetch the object. If fetching succeeds, then the cloud server will copy the relevant fields to the application-level micro-object, after which its

```
void get_msg(bfer_t *p_bfer_message, bfer_t bfer_radix_token)
{
  mo_t   mo;
    :
    :  /* Code omitted for clarity. */
    :  /* Full code in Appendix A. */
    :
  /* Allocate ADTs. */
  mo_alloc(&mo);
    :
    :
  tken_from_radix64_bfer(&tken, bfer_radix_token);
    :
    :
  /* Get a copy, get the payload, copy its buffer.
   */
  mo_create_copy(&mo, tken, psec, csec);
  mo_getref_plod(&r_plod, mo);
  plod_get_bfer(p_bfer_message, *r_plod);

  /* Deallocate ADTs (in random order). */
  mo_free(&mo);
}
```

Figure 3.10: Fetching and reading a "Hello World!" object.

payload can be retrieved.

Again, note the difference with other distributed frameworks. No messages are sent to remote data objects. No remote procedure calls take place: the system only copies requested data from a (possible) remote location.

3.6.4 Example Extension

The following extensions to the example above change the micro-object into a chat channel for IM. Calling the function `do_chat()`, shown in Figure 3.11, will start the exchange of lines of text, using the previously created "Hello World!" object. This function reads lines from the user's terminal, transforms each line to a new `mo_line` micro-object and adds this new micro-object to (the cluster of) the `mo_channel` micro-object. Before entering the read--add loop, the replication policy of `mo_channel` is set to FLOODING, to stress the importance of fast delivery of newly added members, and `mo_get_cluster()` is called with the callback function `fprintmo()`. This latter function prints the payload of the given (token of a) micro-object. Once it has been set as a callback function, using `mo_cter_clbk()` every time a new member is added to the cluster, it will be printed. From the application programmer's perspective, calling `mo_cter_clbk()` is similar to starting a new thread that receives all the tokens as they are

```
void do_chat(bfer_t bfer_radix_mo_channel)
{
  /* Declare local variables.
  */
  mo_t   mo_line, ... ;
    :
    :  /* Code omitted for clarity. */
    :  /* Full code in Appendix A. */
    :
  /* Allocate ADTs.
  */
  mo_alloc(&mo_line);
    :
    :
  /* Set replication of the channel (object) to flooding.
  */
  mo_put_replication(&mo_channel, FLOODING);
    :
    :
  /* Start forwarding the incoming tokens.
  */
  mo_put_cter_clbk(&mo_channel, &cter_tracker, fprintmo, stdout);

  /* Start the line with a prompt.
  */
  strcpy(line, "1>> ");

  /* Direct all input lines to channel.
  */
  while(fgets(line + 4, sizeof(line) - 4, stdin)) {  /* Read line. */
    plod_put_string(&plod, line);                    /* In payload. */
    xpir_uinc(&xpir);                                /* Up count. */
    mo_create_new(&mo_line, xpir, plod, psec, csec); /* New line-MO. */
    mo_cter_add_mo(&mo_channel, mo_line);            /* Append. */
  }

  /* User typed ^D: Stop forwarding the incoming tokens.
  */
  mo_put_cter_clbk(&mo_channel, NULL, NULL, NULL);

  /* Deallocate ADTs.
  */
  mo_free(&mo_line;
    :
    :
}
```

Figure 3.11: Using micro-objects for instant messaging.

added to the cluster of the micro-object. To make the lib-server thread stop calling `fprintmo()`, the `NULL` argument is used for `mo_cter_clbk()`.

As an alternative, Figure 3.12 shows a single-threaded, busy-loop version, using nonblocking calls only. This version is almost identical, but now, the

```
void busy_chat(bfer_t bfer_radix_mo_channel)
{
  /* Declare local variables.
  */
  mo_t   mo_line, ...;
    :
    :  /* Code omitted for clarity. */
    :  /* Full code in Appendix A. */
    :
  /* Allocate ADTs.
  */
  mo_alloc(&mo_line);
    :
    :
  /* Start a line with a prompt.
  */
  strcpy(line, "3>> ");
    :
    :
  /* Loop until there is no more input.
  */
  for(;;) {
    /* Print cluster so far.
    */
    mo_getref_cter(&r_cter, mo_channel);
    pr_cluster(mo_channel, *r_cter);

    /* Get user input, but don't wait to long.
    */
    if (!read_noblock(line + 4, sizeof(line) - 4)) break;

    /* If there is new data, put it on the channel.
    */
    if ('\0' != line[4]) {                          /* Read line */
      plod_put_string(&plod, line);                 /* 2 payload. */
      xpir_uinc(&xpir);                             /* Up count. */
      mo_create_new(&mo_line, xpir, plod, psec, csec); /* Create MO. */
      mo_cter_add_mo(&mo_channel, mo_line);         /* Append. */
    }
  }
    :
    :
  /* Deallocate ADTs.
  */
  mo_free(&mo_line);
    :
    :
}
```

Figure 3.12: Single threaded version of code shown in Figure 3.11.

cluster is retrieved and printed inside the loop replacing the two `mo_cltr_clbk()` calls bracketing the loop. Note that these two examples are compatible: they can be run by different applications on different machines and actually provide

the exchange of chat lines. This compatibility is due to both programs (1) using the MO system and (2) agreeing on how micro-objects are clustered.

Also note that the "Hello World!" micro-object that is used as a channel-DAO (Distributed Application Object), would normally contain a symmetric key to be used like a session key for the channel.

3.7 The Micro-Object Clusters

This section demonstrates how micro-object clusters can be used to construct a complex fully mutable DAO. In the MO system, an application (programmer) defines every DAO as a *single* micro-object with a (application-specific) graph structure. Sharing only a representative micro-object will nevertheless enable distributed applications to share a multitude of objects as these other objects can be organized in a graph of any shape. Usually the graph structure is a tree, and the micro-object representing the DAO is the root of the tree. This allows convenient access to the entire tree from just one token. Also, a tree shape explicitly allows a DAO to consist of other sub DAOs. More specifically, if every DAO is represented by one micro-object, complex DAOs can be crafted by creating a micro-object and adding (sub) DAOs to its cluster.

As an example, consider the realization of a *file DAO* shared by a number of distributed file-system applications, shown in Figure 3.13. The cluster of the file DAO, F, contains two (tokens of) block DAOs. The cluster of the first block DAO, B1, contains three content DAOs. The second block DAO, B2, holds two content DAOs. The content of a file DAO is defined as the concatenation of the content of its clustered block DAOs, ordered by expiration date (B1, B2 in this case). The content of a block DAO is defined as (the payload of) the last content DAO from its cluster, ordered by expiration date. Thus the content of file F is the payload of C3 followed by the payload of C5.

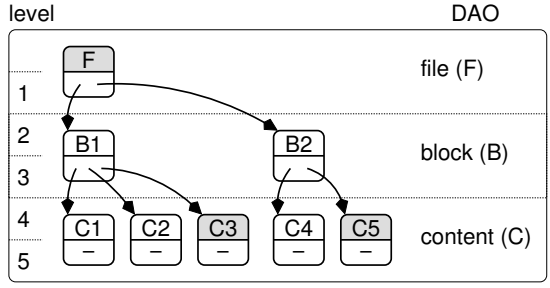

Figure 3.13: The realization of a distributed file of two blocks.

From the MO system point of view, every DAO is a regular micro-object and the structure of the graph originating from its cluster has no meaning to the MO system. One of the unique features of the MO system, is that it still utilizes these graphs for grouping micro-objects. Grouping can be used to improve the effectiveness of replication, especially if the objects are small. This phenomenon is also known from other fields, such as data clustering for efficient replication and distribution of databases [49]. By using a *replication level* indicator in combination with a DAO, an application can generically inform the MO system that a replication policy should be applied to all micro-objects in a subgraph originating from a given cluster. Replication level 1, for example, indicates that only the file cluster itself should be replicated.

To continue this example, assume that the sharing applications have set the replication policy of their copy of F to FLOODING at level 4. Consider what happens after one of the applications changes (the second block of) its local copy of the file. To update the file, a new block, C6, is constructed to replace C5. Next C6 is added to (the cluster of) the local copy of B2. Due to the flooding level of F, B2 (level 2 and 3) and the payload of C6 (level 4) will be flooded, too. Had the level been set to 3, only the change would have been flooded (i.e., the cluster of B2), but not the payload of C6. Obviously, setting the level to 5 or higher, would not have made a difference. Note that *the file DAO is fully mutable*, even though micro-objects are not. Note that the above example is oversimplified, a more realistic and slightly complexer example would be based on journaling.

The replication level is thus seen to provide the application (programmer) a simple yet powerful means to express replication of larger groups of micro-objects.

3.7.1 Managing Clusters

The concept of one cluster per micro-object is a simple one. The cluster of a micro-object contains only tokens and tokens can only be added to a cluster, not deleted or changed. However desirable the simplicity of this scheme might be, it should also be possible to create an efficient implementation. An in-depth description of the implementation is not part of this thesis. There are many details, and for the interested reader, the source code of the micro-object system will be a satisfying read. However, due to the central role clusters play in the design of the MO system, the disclosure of some implementation details is warranted at this point. Probably the most pressing questions are, first, "Can the implementation handle large clusters efficiently?" and second, "Can the

implementation handle frequent cluster additions efficiently?"

Cluster Growth

The design of the MO system does not allow the application to actively remove tokens from a cluster at the MO system level. At the application level a DAO can be arbitrarily changed by adding micro-objects that carry instructions for any change. In fact, adding any type of "delete" operator for clusters at the MO system level would not only lead to a more complex interface, but it would also introduce novel types of failure. For example, imagine one application process that first adds a token to a cluster and then subsequently removes it again. Without additional and hard-to-keep guarantees, a change in the order of these instructions could lead to all kinds of subtle application failures. Without a delete operation, it might seem that all clusters are doomed to keep growing indefinitely. However, since each cluster is part of one unique micro-object, a cluster will not live forever, since it expires when its micro-object expires.

Still, if a cluster gets really big, it could consume a significant amount of resources. As with any burden on system resources, it is up to the application programmer to make the trade-offs involved. That said, there is a mechanism that allows any cloud server and lib-server to locally decide when a token can be removed from a cluster. Since micro-objects expire, it is no use hanging on to a token in a cluster if its corresponding micro-object has expired. Since the expiration date is part of the token, any cloud server can locally decide when a token can be deleted from a cluster.

For clusters containing thousands of tokens, the programmer has two ways of easing the burden on the system. A well-constructed set of application programs can make sure large clusters either belong to a short-lived micro-object, or is mainly populated by tokens of short-lived micro-objects. Still, a misbehaving or ill-designed application can easily eat away a lot of system resources, but that is not a problem that the MO system tries to solve. A similar problem exists with disk space. However, every individual cloud server is free to implement local limits on resource usage, just like a file system can feature a quota system.

In short, the MO system does support, but does not enforce, resource-friendly applications, but local quota systems could be added.

Cluster Synchronization

If a micro-object's cluster gets updated frequently, it would be very uneconomical to bluntly send the whole cluster to other servers upon every token addition. Clearly this is not a resource-friendly way to synchronize an ordered list of to-

kens, that is a cluster. Most desirable would be if every cloud server would send only those updates to other cloud servers that they need. This, however, implies that all cloud servers would need to keep track of the state of all other cloud servers (and know what they need). Not only does this put a huge dent in storage resources, it also introduces massive opportunity for partial failures to wreak havoc. In fact, such a method would introduce into the MO system almost all the problems other distributed-programming frameworks have. For this very reason the MO system is designed to maximize local decision making, that is it is designed to minimize knowledge of remote state. The method described below does data synchronization without the need for state synchronization.

One particularly hard problem to overcome is state changes *during* data synchronization. In a multiparty system, *a data synchronization mechanism should be indifferent to state changes during updates.* Given the limited capabilities of micro-object clusters, there exists a relatively simple and efficient way of stateless synchronization. This synchronization is based on the exchange of update messages. All the necessary state information is carried inside each individual update message. No state of remote hosts needs to be kept and no state in between update messages needs to be locked. Hence, interpretation of incoming update messages as well as the construction of response update messages is based upon the local state and the incoming message only.

The basic principal of cluster updates is the notion that total synchronization does not need to be achieved with the exchange of only one update message or one pair of messages or any predetermined number of messages. As long as every update message can only contribute towards synchronization but not hinder it, total synchronization becomes a matter of exchanging enough messages. The MO system features eventually consistent clusters. Eventual consistency is not cure for everything but it is important to a whole class of distributed systems including unified messaging. To do this, an update message needs to contain a description of the content of a cluster. From now on, "the MO system's description of the content of a cluster for update messages" as described below, will be abbreviated to *content list*. A remote content list can be compared to the local cluster content, yielding several of three conclusions. One, the remote cluster contains a token that needs to be added to the local cluster. Two, the local cluster contains a token that needs to be sent to the remote site. Three, a mismatch exists, but there is not enough information to fix it. A typical outcome of comparing a remote content list with a local cluster might be that three new tokens are added to the local cluster, one token in the local cluster needs to be sent to the remote site, and one mismatch that needs further synchronization to solve. Based on this outcome, the three new tokens are added to the local

cluster and a new content list is constructed and sent out, containing the one token that the remote site was missing, and *more specific* information on the unsolvable mismatch. To the remote site, this is not an answer to the previous content list, but an independent message to further synchronization.

Example of Content List Driven Synchronization

As a simple example, let's consider two servers (A and B) that need to synchronize the cluster of their respective local micro-object copies, as shown in Figure 3.14. Let us further assume they each have four tokens, as shown in Figure 3.14(a). In Figures 3.14 and 3.15, for simplicity, the expiration date will be represented by a single-digit number. Analogously to the ordering of tokens on expiration date, the tokens in the examples below are ordered by this fictional number. At some point in time, some local application adds a token (token 4)

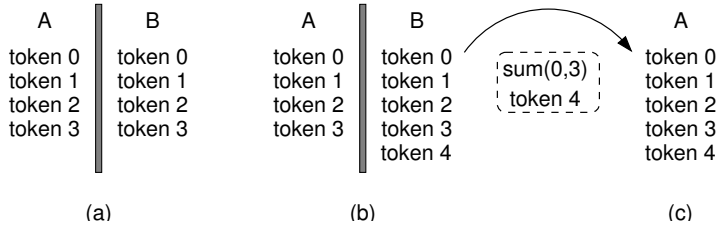

Figure 3.14: Syncing two clusters after insert of token, using a content list.

to the cluster at server B, as shown in Figure 3.14(b). Let us assume server B has to enforce a replication policy that forces it to try to synchronize with A. So it sends out an update message containing two items, first, a checksum of the older tokens, second, (a copy of) the new token. Note that server B does not anticipate an answer nor does it need to keep track of update messages that are sent out. That is, B does not keep a communication history. Upon reception, server A examines the update message item by item. First, it finds the checksum for the first four tokens, it verifies the checksum, and since it matches the known tokens, it does nothing. Next it finds the new token (token 4), and since it does not have this token in the local copy of the content list, it adds the new token to its list. Now A arrives at the end of the update message. Since there are no tokens known to server A that were not in the update message there is no need to send back anything, and synchronization has been accomplished, as shown in Figure 3.14(c).

Please note the low overhead. Even if the initial situation had contained

one thousand nodes instead of just four, there was only one update message sent that is only marginally larger than the raw size of the new token. On the Internet, one UDP (User Datagram Protocol) packet could suffice. Note that a replication policy *might* or might not force server A to acknowledge the synchronization has finished, or it might force server A to use the less error-prone TCP protocol. However, as with any acknowledgment in communication, there has to be a last acknowledgment, and therefore there is not ever going to be a way for both parties to know for sure if synchronization succeeded. Tanenbaum et al. calls this the "two army problem" [61]. There is but one guarantee, if server A and server B keep communicating, synchronization will become ever more probable. This also holds if more than two servers try to synchronize

If server A initially had been missing a token (e.g., token 2) in Figure 3.14, the synchronization after adding token 4 would have proceeded differently. Initially server B would send the same update message as it did in the previous example, as shown in Figure 3.15(a). Upon reception, A would start comparing

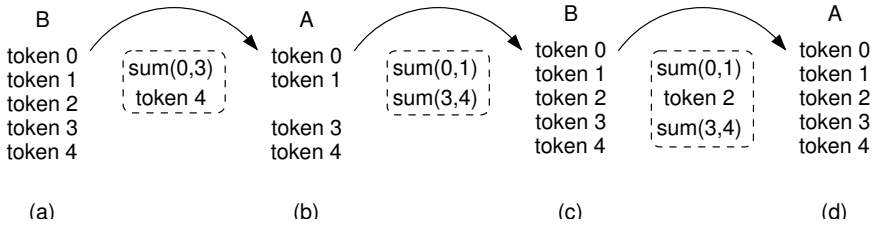

Figure 3.15: Syncing two clusters with more than one token missing.

the items in the message. The first item would be a mismatch, and A would know something was out of sync between number 0 and 3. What exactly is missing cannot be inferred from the update message. So after processing the other items from the update message (adding token 4 in the process), A would *split the sum* for tokens 0, 1, 3, and 4, and it would send that back to B, as shown in Figure 3.15(b). How A splits the sum is discussed below. From that server B could determine what A was missing. A new update message would be constructed, containing the missing token (token 2) and two checksums. Upon reception, A would process it and add the missing token to its cluster. Again, note first, how efficient this algorithm is in this case and second, that each packet contains all the information needed to process it properly (i.e., every update can be processed individually) independently of remote state or even local communication history.

There are two obvious parameters that influence the performance of this

algorithm. First, the number of consecutive tokens that will not be replaced by their checksum. Let's call this replacement value "R." Second, the number of parts a mismatching checksum is broken into. Let's call this fragmentation value "F." Both R and F can have a huge impact on the total number of bytes exchanged (#B) and the total number of update messages exchanged (#M).

The Replacement Value R

R holds the number of consecutive tokens that will not be replaced by their checksum. For example, with a replacement value, R of 1, any number of consecutive tokens will be replaced with their checksum. With R equals 10, any number of consecutive tokens over 10 will be replaced with their checksum and any group of tokens less than or equal to 10 will be inserted in the update message verbatim. Parameter R could easily be set to 100 or above, the effect would be that most update messages would grow in size (i.e. #B would grow) but #M, the number of update messages needed for a full synchronization, might go down.

Now the question rises how well this implementation does on arbitrary synchronization needs. The answer is, in general, not too well, especially if added tokens have random expiration dates. However, if added tokens in clusters group around one date, the algorithm is highly efficient. To demonstrate this, the results of two simple tests (test A and B) can be compared. In both tests a cluster of 990 tokens is updated by adding 10 new tokens. The number of messages and the number of bytes needed to propagate these changes are recorded for both tests A and B. The only difference between the tests is the position of the new tokens. For test A, tokens are inserted at the end of the cluster. For test B, tokens are inserted at random positions. These tests simulate a situation where one local copy of a micro-object is missing 1% of its clustered tokens. The clusters of these local objects are then synchronized. The cloud server holding the smaller cluster sends its cluster to the cloud server holding the complete cluster. Due to base replication, the receiving cloud server replies. The two cloud servers keep exchanging messages until both clusters contain the same 1000 tokens. The results are displayed in Figure 3.16. The starred area (A) represents the number of update messages exchanged (#M) when the added tokens group at one extreme of the cluster. The minimum number of messages is two because the first message will show what the smallest content list is missing and the second message will contain all the missing tokens. The striped area (B) represents the total number of update messages exchanged if the 1% new tokens do not group at the end of the token list, but are randomly dispersed between the other 990. As shown, #M is ranging from 12 to the minimum number of

messages with increased values of R.

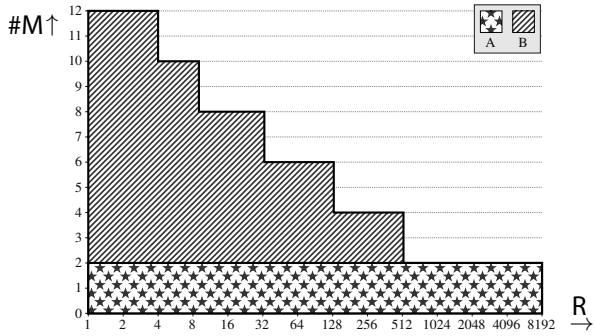

Figure 3.16: The number of messages (#M) needed to synchronize a cluster of size 1000 against parameter R. Note that only two messages are needed when the changes group (area A).

Figure 3.17 shows a similar graph for #B, the total number of bytes exchanged. Note that the maximum for #B is the sum raw size of the smallest and biggest clusters, plus some bytes for overhead. The starred area (A) represents

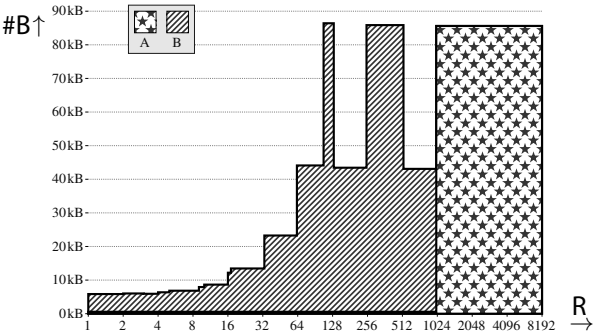

Figure 3.17: The number of bytes exchanged (#B) to synchronize a cluster of size 1000 against parameter R. Note the characteristic block wave pattern for R over 63. Also note that if R >= 900 the number of bytes exchanged maxes out.

the number of bytes exchanged if the new tokens group at one extreme of the cluster. In fact, the starred area is not very visible on the left hand side, due to the scale of the y-axis. The number of bytes is rather constant at (merely) 600

bytes, until R reaches 990. The striped area (B) represents the total number of bytes exchanged if the 1% new tokens do not group at the end of the list. These graphs clearly show the influence of R on #M and #B if updates to the cluster do not group. Smaller values of R cause an increase of #M, bigger values of R inflate #B. Past a certain value of R (63 for size 1000) the number of bytes exchanged starts oscillating roughly, between the raw size of the large cluster and twice that. Because at some values for R every update message but the last contains checksums only, leaving #B at about the raw size of the cluster. Yet for other values or R, the last two messages both contain all the tokens in full, leaving #B at about twice the raw size of the cluster. Note that two parties synchronizing do not need to have the same value for R or even know the other party's value, each party can locally take decisions on what value for R to use, or even change it for every new message. This should not come as a surprise given the near statelessness of the whole synchronization process.

The most important observations are these. First, if cluster updates do group, R does not have much of an influence at all, as long as R stays well below the cluster size. Second, exchanging content-lists is really efficient compared to a brute-force approach.

All these observations still hold if the clusters are increased by several orders of magnitude. For example, if the experiment is repeated with a cluster

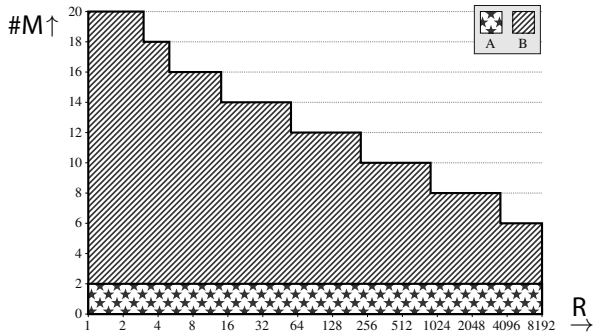

Figure 3.18: The number of messages (#M) against parameter R with a cluster size of 100.000.

size of 100.000, the graphs show a similar trend, as shown in Figure 3.18 and Figure 3.19.

Since it is plausible that most applications can assure that added tokens can have increasing expiration dates, further research in the effects of parameter R falls outside the context of this thesis.

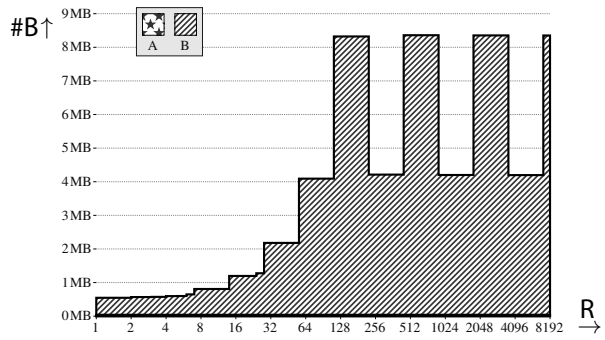

Figure 3.19: The number of bytes #B against parameter R for a cluster of size 100.000. When newly added tokens group (area A), #B is constant around 44kB.

The Fragmentation Value F

F holds the number of parts a mismatching checksum is broken into. The fragmentation value, F, plays a role when a group of tokens is represented by a checksum in an update message. If the checksum does not match the checksum over the same group of tokens in the local copy of the cluster, there is a mismatch. An example of this was given in Figure 3.15, where server B sends a sum of the tokens 0 through 3 to server A. Since server A does not have token2, there is a mismatch. With F set to 2, the group of tokens 0 through 3 is split in two. Had F been 4, the group would have been split in four, effectively sending tokens 0 through 3 individually. Note that as with the replacement value above, F can be dynamically changed and does not have to be communicated to remote parties.

The value of F can be researched for its effect on the number of messages #M and total number of bytes #B in a similar fashion as the replacement value above. A similar set of graphs can be drawn up. The graphs plotting F against the number of messages #M (Figure 3.20(a) and 3.21(a)) resemble the corresponding graphs for R (Figure 3.16 and 3.18). The graphs plotting F against the total number of bytes #B (Figure 3.20(b) and 3.21(b)) resemble a clipped sine wave for larger values of F, just as the the corresponding graphs for R (Figure 3.17 and 3.19) show a block wave. Note that a fragmentation value of 1 is invalid, because it would signify that a mismatching checksum would be "split" into one new checksum. Hence there is no value for 1 in the graphs of Figure 3.20 and 3.21. The main observation, however, is similar to the conclusion for the

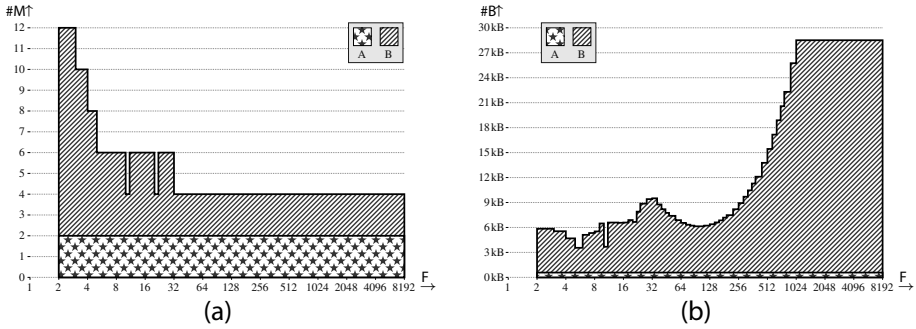

Figure 3.20: Plots for cluster size 1000: (a) number of messages (#M) against parameter F, (b) number of bytes (#B) against parameter F.

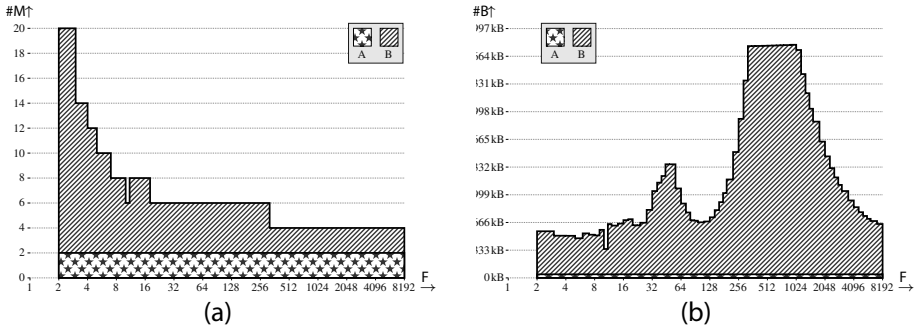

Figure 3.21: Plots for cluster size 100.000: (a) number of messages (#M) against parameter F, (b) number of bytes (#B) against parameter F.

replacement value above, the fragmentation value F has shown to have hardly any effect if added tokens group.

Note that F was fixed at 2 for the graphs plotting #M and #B against R. Even so, R was fixed at 1 for the graphs of the fragmentation value. If newly added tokens cluster, increasing both R and F at the same time has shown to have a similar minimal effect, on #M and #B. Using dynamic values for R and F is also possible, but given that the effect is almost exclusively significant when added tokens do not cluster, no further research on the effect of different values is

warranted at this point.

3.8 Security and Emergence

This chapter introduced a very different approach to distributed computing. Instead of sending messages to (possibly replicated) remote objects, the micro-object system forces operations to take place on local copies only, keeps data in objects immutable, and supports only local graph-like data structures from which objects can never be removed. There are several ramifications of this approach that have been barely touched upon. Below a brief discussion of two important ones: security and emergence.

3.8.1 Security

Building a secure large-scale distributed system requires that the security infrastructure is integrated into the design from the start. Therefore, the security infrastructure is natively incorporated into the MO system, as illustrated in Figure 3.1. The MO system needs data security and system security. *Data security* is there to protect micro-objects from unauthorized access, but also to protect applications against bogus micro-objects. *System security* concentrates on serving benign applications, while denying service to malicious applications.

Data security

For data security, the MO system provides separated security policies that utilize (but are not limited to) end-to-end encryption and authentication. All sensitive security data appears in plain text only in the application address space.

Bogus micro-objects can be detected by end-to-end authentication. However, a bogus micro-object will be detected only at the highest (i.e., application) level. Therefore, bogus micro-objects still threaten the functionality of the lower level (i.e., the MO system itself). To deal with DoS (Denial of Service) attacks, the MO system has been designed such that most bogus data can be detected early. Note that it is quite easy to generate a bogus micro-object and then calculate the correct hash value for its token. However, it is unlikely that some application would ever request such a micro-object. Generating a bogus micro-object in response to a specific request is computationally much harder, because the hash of the requested micro-object is given as part of the request.

System security

The system security of the MO system is still largely unexplored. The MO system has rudimentary protection against abuse of storage and transport. In principle, cloud servers can be tricked in to storing bogus data, but it will end up in the cache so that the harm is limited. A set of cloud servers can sometimes be tricked into transporting bogus data. However, newly developed policies and security for replication might remedy this. The MO system does not yet have protection (other than its hot-spot handling), against denial of service. For example, a flooding attack will put parts of the MO system out of function. Also, the MO system suffers from the security bootstrapping problem: in order to set up secure communication between two given parties, some pre-existing shared secret is needed. Flooding and bootstrapping are common security problems, and they are not specific to the MO system nor is it clear that these problems can be solved by changing the design.

As mentioned in Section 3.5.5, the MO system does not (yet) possess any data-flow shielding, and may thus leak sensitive data. However, special replication schemes could be devised to make traffic analysis more difficult.

3.8.2 Emergent Behavior

The emphasis on local decision making has important ramifications for overall system behavior. For example, as explained above, objects can be replicated across the system only if local policies of initiating and intended peers match. In contrast, replication in virtually all traditional distributed systems is based on explicit and centralized control. The effect of having only local policies leads to emergent behavior, observed as the flow of (copies of) micro-objects between servers.

It remains to be seen to what extent this emergent behavior can actually be controlled. One avenue that will have to be further explored is developing various replication policies and to see how combinations affect the replication and distribution of micro-objects. Although the loss of centralized control can be seen as a disadvantage, local decision making simplifies development and will certainly lead to much better scalable solutions for some problem domains.

In this light, this approach is to be compared to the recent increase in gossip-based solutions, which all evolve around local decision making [27]. These solutions have in common that only by fine tuning local decision rules can one observe desirable global behavior. Unfortunately, the relation between this local tuning and global behavior is often not well understood, and only recently have studies been published in which different approaches are systematically

compared [33]. However, it is also clear that local decision making has excellent scalability properties, allowing systems to easily grow to millions of nodes. This point has already been demonstrated by traditional decentralized systems such as those for exchanging news and E-mail.

3.9 Related Work

The main argument in this chapter is not new: *distribution transparency is a bad idea.* This research offers an alternative to transparency, making distribution visible to the programmer in a manageable way. The crux to this approach is to introduce distribution at a lower level, utilizing the lower complexity that comes with it. In some way DSM (Distributed Shared Memory) takes a similar approach; Tackling the distribution problems at a low level. However, I argue that DSM does not concentrate the effects of partial failures in a few manageable places, but spreads the problem evenly throughout the whole program. Any DSM usage becomes a possible point of failure.

This especially goes for totally transparent DSM, where the operating system integrates shared memory with the virtual memory management. With totally transparent DSM every memory access can fail, turning even a basic instruction like "add one to variable X" into a liability. But even with function-call driven DSM, where the application has to request blocks of memory explicitly, every request turns into a potential problem. One could argue that micro-objects are handles to blocks of shared memory. In effect, this is true, but micro-objects have additional properties. First, they are immutable and second they feature a generic way of relating to each other: clustering. The first property, immutability, severely simplifies distribution. The second property, clustering, allows the support system to better predict the needs of an application, allowing dynamic, application-specific replication. In other words where the distributed-application-object approach is too high level, the DSM approach is too low level.

In Section 6.2 a detailed comparison is made between existing distributed frameworks and the micro-objects framework.

3.10 Summary

Current message-to-object based distributed frameworks ignore or fail to adequately address partial failures. For distributed systems based on small, highly reliable networks, and error ignorance, as for centralized solutions, this might

lead to acceptable behavior. However, for most large-scale distributed applications, this leads to unacceptable emergent behavior. As an alternative, the micro-object system is simple and clean. It makes local decision based solely on information received, not on the suspected state of a remote party. Therefore, remote sites never have to acknowledge reception of information, so the micro-object system does not suffer from partial failures. If an application needs some remote information, it can only request it using an API call. These API requests are clearly recognizable as blocking calls to the application (programmer) to prevent any surprises. The design and implementation indicate that this model is relatively simple to realize. However, it is yet too soon to draw hard conclusions on the viability of the micro-object system, although it is clear that it contains the essential elements to tackle the hard problems that have been hampering large-scale distributed systems. Some of these hard problems, notably handling partial failures, are strongly alleviated by the choice of combining local computing and immutability. The drawback is loss of distribution transparency but also loss of the partial failure that comes with it. It is clear this is only the beginning of exploring this new paradigm.

❧ *"How to implement unified messaging atop a cloud of micro-objects?"* To answer that question, I sketched out a design in micro-object graphs. I showed it to Jeroen Snoeij, a computer-science master student after I explained both unified messaging and micro-objects. *"Would you like to implement unified messaging using micro-objects?"* I asked him. The discussion that followed ended half a year later when he finished his master thesis: *Unified Messaging with Micro Objects*. Though more a proof of concept than an industrial-strength application, it proved that the two-layered approach was viable. When I asked him if micro-objects had made it easier to implement unified messaging. His response was: *"Micro-objects not just made it easier, it would have been impossible without."* After hearing his answer I made my promotor smile by claiming *"Cut, print, it's a wrap."* Little did I know. His smile meant I still had to spend quite some time on the important question that followed it. *"How about performance and performance/resource trade-offs?"*

Chapter 4

A Unified Messaging System

> "Most people who have worked on this sort of thing say the same; it looks deceptively easy till you actually get in to try and do it."
>
> *Eric Allman*, the original author of sendmail.

By using the micro-object middleware layer described in Chapter 3, it should be somewhat more straightforward to implement large-scale distributed messaging systems. Primarily, because the MO middleware allows *direct addressing of data* instead of indirect addressing of data at network end points. This frees the UMS from handling the actual locating and transporting of messages. This chapter will focus on a large-scale, MO based, unified messaging system. In this system, as with any large-scale distributed system based on micro-objects:

> *A distributed-application object (DAO) is a rooted weakly connected graph of micro-objects, represented by one root micro-object from that graph.*

From this, it follows that integrating several basic DAOs into one new more complex or more powerful DAO is a simple matter of creating a new micro-object and putting the basic DAOs in its cluster, as shown in Figure 4.1. Representing a DAO by one (root) micro-object allows for a hierarchical build up of DAOs. Though a DAO is built up from micro-objects, it can be seen at another level of abstraction as being built up from (simpler) DAOs. This is one reason why a huge tree of micro-objects can feature enough (layered) abstraction to remain usable, even when distributed over many computers. As

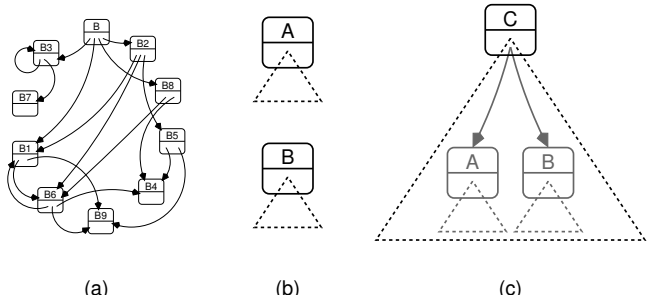

Figure 4.1: Distributed application object composition. (a) A rooted weakly connected graph of micro-objects forming one DAO with root B representing it (b) Two micro-objects named A and B each representing a DAO. (c) One new DAO, C, is created by adding A and B to the cluster of a new micro-object.

will become apparent below, a large messaging system could feature billions of connected micro-objects, distributed over millions of computers, yet every level of abstraction can be understood in terms of underlying abstractions. Needless to say that this comes in handy when designing and implementing such systems.

To gain practical experience and as a testing ground for micro-objects, two messaging related messaging systems were implemented. First, a plugin for an existing IM client was adapted to use micro-objects, second a UMS proof-of-concept implementation was written.

The IM client

The world of instant messaging is partitioned into several incompatible systems. Leaving aside the proprietary instant messaging systems provided by network operators and focusing on popular instant messaging systems on top of the Internet, we find (in alphabetical order) AIM, Bonjour, Gadu-Gadu, Google Talk, Groupwise, ICQ, IRC, MSN, MySpaceIM, QQ, SILC, SIMPLE, Sametime, Skype, XMPP, Yahoo!, and Zephyr. This list is by no means complete. Clients of one IM system can usually not communicate with clients of another. There is, however, one very popular IM client that can communicate with all these IM systems: Pidgin. Pidgin supports what is known as a plugin. A plugin allows a generic IM client to interface with a specific IM protocol. By providing a plugin based on the MO system, a full-blown instant messaging system using micro-objects can be implemented. This plugin has two parts. One part implements sockets on top of micro-objects, the other part implements the administrative

management of the plugin. At first it might seem strange that a socket interface would be put atop of micro-objects which in turn are put atop sockets. However, there are two main reasons why this might be considered an interesting experiment. First, the socket interface supports generic communication, not just traditional end-to-end IP communication. Supporting it could make porting IP-based applications to MO based ones much easier. Second, this setup allows a traditionally designed Internet messaging client to talk to other clients that sit directly on top of micro-objects. For an example of the latter, with the plugin, Pidgin is perfectly capable of communicating with both IM example implementations as given in the Section 3 in Figures 3.12 and 3.11.

So, the MO-based socket implementation allows applications that traditionally run on an IP socket to run on the MO layer. Given the security and replication policies of the MO system, this allows the MO-based socket implementation to offer additional services like history, low latency, and security, to existing distributed applications almost for free.

The UMS implementation

A much bigger messaging system has also been implemented. For practical reasons we took two measures. First, readily available code was used as much as possible for the GUI (Graphical User Interface) and the data storage. Second, not all types of messaging were supported, only rudimentary E-mail, USENET News, and IM style messaging. Below is a description of the code that was available beforehand. The design allowed for any standard browser client to function as the GUI of the UMS. Since a browser client necessitates a client/server implementation, a *UMS server* was created as a stack, with a simple web server on top and some message-handling software below that, as shown Figure 4.2. Message storage is handled by a lightweight SQL (Structured Query Language) library, called SQLite. Message exchange is handled, of course, by the MO system. So the lion's share part of the UMS system consists of "prefab" code. All that remained was designing and implementing a UMS library to glue the upper layers (browser and web server) and the lower layers (data management and message exchange) together. This chapter will focus on the new code written for the UMS.

4.1 Implementation Description

Clearly the proof-of-concept implementation was never meant to be a replacement for all current messaging systems. As stated above, it aims only at mim-

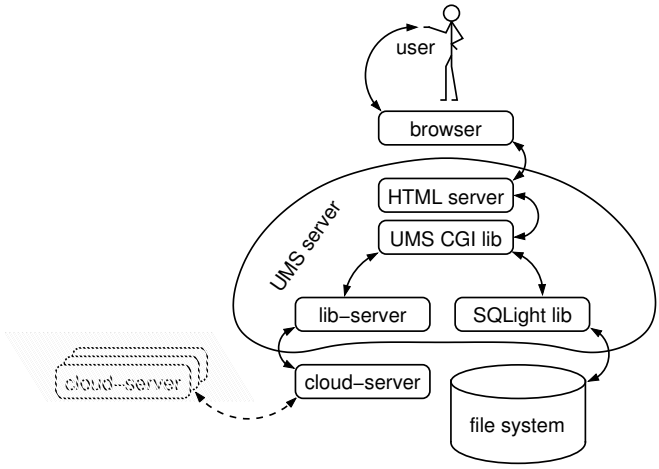

Figure 4.2: A simplified, client-server based, UMS system. The UMS client (a standard web browser) provides the GUI. The UMS server consists of a web server, a UMS library that uses a SQLite library to store and a MO lib-server to exchange messages.

icking the basic behavior of E-mail, USENET News, and IM. Though this is not truly unified messaging, it does cover the major classes of popular current messaging systems. The description below is based on the proof-of-concept implementation. Hence it is not a description of a complete and unified messaging system, yet it tries to demonstrate an easier to understand messaging system hinting at unification. The last section of this chapter describes what additions would be needed in order to create a system that would be more capable of true unified messaging.

The next sections describe the build up of the UMS target as a DAO, as discussed in Section 2.2.2. The section after that does the same for the TISM distributed-application object.

4.1.1 Target Design

In unified messaging the target is used as a replacement for message destinations mimicking the E-mail *inbox*, the IM *session* or *channel*, and the USENET *newsgroup*, all in one. Note that these three message stores have a lot in common and the naming differences signify mainly conceptual differences. The design of the target is a rather straightforward message store, with a user-selectable name and a set of messages. However, to accommodate the IM "presence" feature,

a presence DAO is added. The resulting design is shown in Figure 4.3. Note

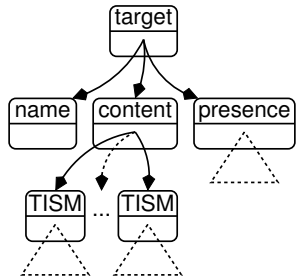

Figure 4.3: The target-DAO in micro-objects. Each target DAO consists of a name DAO, content DAO, and presence DAO. Micro-objects are depicted as rounded rectangles and DAO's are rooted graphs with their root micro-objects labeled. For example the content DAO is the whole graph below the micro-object labeled "content."

that the name of the target is in a separate DAO. This allows for easy renaming. However, renaming was not implemented.

The content of a target is a set of plain TISMs (see below). No subtargets are allowed in this implementation, neither in the content DAO nor in some other part. Though apparently straightforward, an important yet slightly less obvious implication of this design is worth mentioning. This design allows a TISM to be part of zero, one or many targets. In the E-mail and USENET system this is called forwarding and cross-posting respectively. There is no equivalent of cross-posting in most IM systems. The presence DAO will be explained below.

With an existing approach to the development of such a distributed system, different replication policies of the content DAO and the presence DAO could lead to separate, and very different implementations. With micro-objects, content and presence are implemented in a similar fashion. Setting the right policy would simply be a matter of choosing the right replication policy for each DAO, matching performance to desired behaviour. This is one of the stronger features of the MO system: multiple, dynamic, per (sub) object, replication policies that share the same implementation within a single application framework.

For example, imagine a user interested in the presence of a target only. Let us assume the user's messaging application would be holding a local copy of a target-DAO like the one in Figure 4.3. The application could set the replication policy for the presence-DAO such that it would be updated regularly while not setting the replication policy on the content-DAO, forgoing even the notification of any chance to the content. At some point the user might indicate to

be interested in the content too. Immediately the user's messaging application could switch on a (different) replication policy on the content-DAO.

For another example, imagine a user's messaging application setting two replication policies on the content sub-DAO. One replication policy is resource hogging but tracks changes closely. The other replication policy is light on the resources but slow. The first replication has a level of one, the second a level of ten. This way, users could be notified of the arrival of a new TISM almost instantly, but the actual TISM itself would trickle in. Note that the MO system will instantly retrieve any TISM the application requests explicitly, independent of any active replication policy.

To summarize the above examples: It is useful for a single target-DAO to have independent replication of its presence and content sub-DAOs to better serve the user's needs. Likewise, it is possible to replicate one DAO using two policies at different levels to create a useful mix of low and high latency.

The presence DAO

The presence DAO contains short-lived status DAOs. If some user wanted his or her presence in this particular target to be known, his or her application should insert status DAOs in the clusters of the corresponding presence DAO, at regular intervals. This is actually more than most IM systems offer. The latter generally offer a systemwide presence, whereas this implementation offers a per-target presence. A user could opt to announce his or her online presence in a target shared with his or her friends and be offline (i.e., not present) in all others. Note this is by design, not by necessity, for any target could be appointed (or constructed) to manage "systemwide" presence information. Though not utilized in this implementation, one status DAO could be shared amongst many presence DAOs.

4.1.2 TISM Design

In unified messaging the TISM is used as a replacement for information exchange mimicking the E-mail *message*, IM *line*, and USENET *article*. As with the target, these three datagrams have a lot in common and the naming differences are chiefly cosmetic. Like the design of the target, the TISM is designed as a straightforward tree of micro-objects, with a body, zero or more attachments and a message TISM tree, as shown in Figure 4.4. Given that micro-objects are relatively small and a message can have an arbitrary size, a TISM body can be split into several parts. Since a TISM can have any number of attachments, a similar construct is used for them, just one layer deeper. Since in some mes-

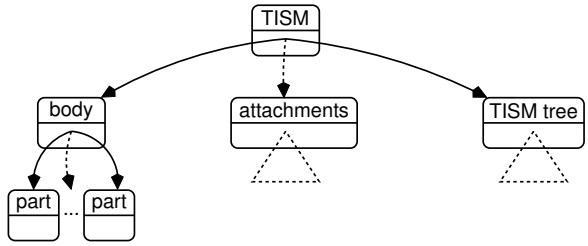

Figure 4.4: The TISM-DAO in micro-objects. Each TISM DAO consists of a body DAO, attachment DAO and tree DAO.

saging systems, i.e., E-mail and USENET, the messages can be related (one message is said to *follow up* another), the TISM contains a DAO that can be used to express these relations. It turns out that a tree structure is well suited for this purpose. First, the body and attachments will be detailed, then the tree DAO will be dealt with.

The design of the TISM DAO is straightforward. However, this layered structure has a distinct advantage with respect to replication. This will be illustrated better with an expansion of the TISM DAO of Figure 4.4, shown again in more depth, in Figure 4.5. As stated in Section 3.7, each micro-object (and

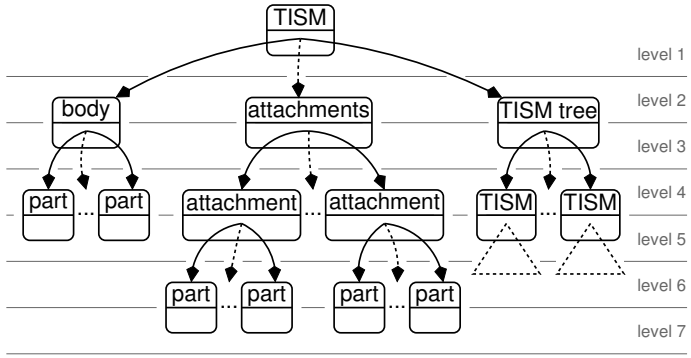

Figure 4.5: A closer look at the TISM-DAO, with replication levels.

hence each DAO) can have its own replication policy and replication depth. Imagine a TISM in a target that is shared between friends. Every TISM in this friend target could get a fast-forwarding replication policy at level 6. There is no need to actually set every TISM policy individually. Setting the replication of the friend target holding all the TISMs to fast-forwarding at level 8 would suffice. This way the content of each part (at level 6) would be rushed to the

recipients the instant they were clustered to any of the micro-objects at level 4/5. Now let's imagine a TISM in some other target that is like an uninteresting news group. The local UMS server could decide to replicate every TISM in this group at level 4. This way, only the main TISM would be rushed to the other side, and if the user ever wanted the attachment, it would have to be retrieved on demand. Even so, replicating at level 4, will make sure all the parts of the body of a message will be rushed to the recipient, but not the parts of the attachments because they were deliberately put in at a lower level. This is a clear example of how the level of replication can influence the design of a DAO. In general, if less important things are put in lower layers they can be excluded from replication by picking a lower replication level.

One way of looking at replication levels is that every micro-object "inherits" the replication policy of (all) its parents, be it at a decreased level. This does not lead to problems if a micro-object happens to simultaneously inherit different replication policies. As discussed in Section 3.5.6, replication is robust with regard to intermediate state change, so one replication policy will never obstruct another.

Clearly the options are endless. For example, it is also possible to combine deep, cheap, but slow replication with shallow, fast and more resource-consuming replication.

The proof-of-concept design of the TISM as shown in Figure 4.5, was designed to allow combining different replication policies with different levels for the root node. However, this is only one, particularly convenient way replication can be done. Finer-grained replication schemes are also possible. For example, looking at a TISM DAO as a combination of three sub-DAOs, (body, attachment, TISM tree, as shown in Figure 4.5) each sub-DAO could have its own independent replication policy, or set of replication policies. For example, a body DAO can be replicated at higher speed and cost, an attachment DAO can be replicated on demand and a TISM tree DAO can be replicated cheaply at a slow pace. Indeed, the replication options seem endless, too.

The TISM tree DAO

Since both E-mail and USENET allow the user to *follow up* one message with several others, a tree-like structure is needed. Followups are usually reactions of one user to a message of another user. Since a followup is a regular message, it can have zero to many followups, too. This leads to tree-like structures, like the one shown in Figure 4.6. The proof-of-concept design was some what more elaborate. The TISM tree did not only allow each TISM to have zero to many *followup* TISMs, thus providing a TISM tree, but also, for efficiency reasons,

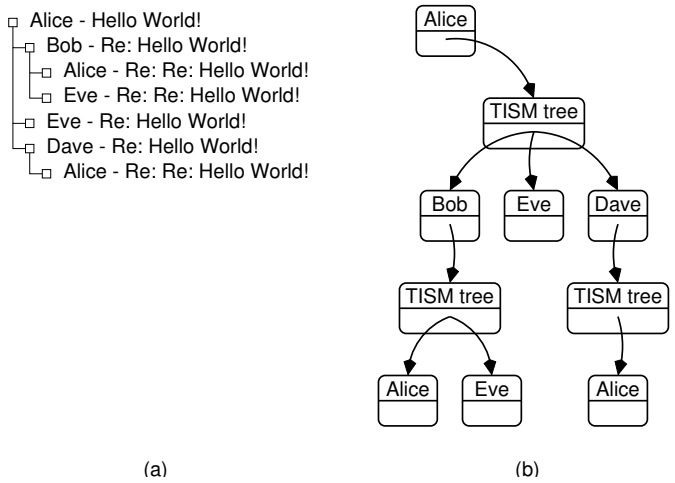

Figure 4.6: Typical followup tree. Alice has posted a message and Bob posted a followup, Alice posted a followup to that and so did Eve. (a) Followup tree as depicted in a typical end-user application. (b) Followup tree as implemented using a TISM tree DAO.

each TISM would hold a link to the root of the tree it belonged to. Note that this would introduce cycles, but also note that the replication strategy can handle cycles, see Section 3.6.2. One of the problems with this design is that a TISM can only be part of one single tree. A truly unified messaging system would also allow for a USENET feature called *cross posting*, by allowing a TISM to be posted in more than one target. To fully combine the *followup* and *cross-post* feature, a TISM should be able to be part of more than one TISM tree. One TISM tree is enough for a proof-of-concept implementation, a more complete UMS would need a slightly different approach. An alternative will be given in the last section of this chapter.

4.2 Security

The most popular versions of E-mail, USENET and IM do not feature much security. Clearly for USENET it would make little sense to encrypt messages. However, given the private nature of E-mail and IM, some protection would be in order. Surely there are many IM and E-mail clients that feature client-to-server or end-to-end encryption. However, popular protocols like Jabber (IM) and SMTP (E-mail), use add-on protocols like, SSL (Secure Sockets Layer)

(client-to-server) and GnuPG (GNU Privacy Guard) (end-to-end) to achieve security. Though this will protect against most eavesdropping, it will not solve other problems, most notably unwanted messages, or spam. The next two sections describe the security as it was designed for the proof-of-concept UMS system. The section after that details about how a messaging system can be protected against spam.

4.2.1 Protecting the Target

As described in Section 2.2.3, every target has a *post-key* and a *read-key*. In Section 2.2 key sharing was introduced as a means to enhance security but also as a means to mimic properties of individual messaging systems. For example, by making a post-key public and keeping the read-key private, an E-mail like functionality was mimicked. However, only the extreme forms of key sharing were mentioned; a key could be either known only by the creator of a target or a key could be public knowledge. In practice, however, a key, like any secret, can be shared by two people or a relatively small group or it can be made publicly known. Figure 4.7 shows a graphical representation of what key distribution belongs to what type of messaging. Clearly for E-mail like functionality, the

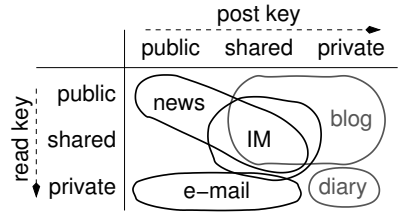

Figure 4.7: The effects of key distribution on the type of messaging service performed.

read-key needs to be private. The post-key should be public or at least shared amongst two people. If both the read-key and the post-key are private, one would have a system that would allow one to write messages only to one self. This area in Figure 4.7 is marked diary. Given the "if you can read it, you can post to it" character of USENET the read-key and post-key are distributed as a pair. As with E-mail, if both keys are kept private it is not really an inter-user messaging system any more. Other than that, any level of distribution for the read-key, post-key pair leads to a news-like messaging system. The IM system largely overlaps with the USENET system. However, if the read-key, post-key pair is shared by too many people, it would be very hard to keep track of who

is saying what, so much so, that it would render the system unusable. As an illustration, Figure 4.7 also holds an area marked blog. It occupies basically the area where the post-key is not known by everybody, and the read-key is not kept private. In short, it occupies the area signifying a messaging system where one or a few users post messages that can be read by many to all other users. Hence the blog label. There might be a good reason to setup a messaging system that allows everybody to post messages (i.e., the post-key is public) but only a few to read them. Hence the open area for a public post-key and a shared read-key. An example would be *customer service:* anyone can file a complaint, and a team of employees (sharing the read-key) can read and handle the complaints. An other example would be a system where a company target allows (many) whistleblowers to complain about employees doing illegal things, but only the CEO can read it. A true UMS system should be able to provide this service, not only to allow customer-service messaging, but also usages not yet thought off. Not being able to anticipate future usage is one of the reasons to create a generic system. Since the proof-of-concept implementation only tries to mimic E-mail, USENET, and IM, this functionality was not implemented.

Note that using asymmetric encryption to mimic messaging paradigms usually leads to a more secure system. For example, with E-mail (without using GnuPG) every MTA passes the message in plain text.

Since the key is not bound to the initiator or their hardware, initiatives (like a newsgroup or a chat channel) can live on even if the original "target creator" stops being involved, simply by transferring the right keys to other users.

4.2.2 Protecting the TISM

Like the target, each individual TISM is protected too. TISMs, however, are protected with symmetric keys, aptly named TISM-keys. At first the proof-of-concept implementation did not have encryption of the TISM DAO. It took some effort to come up with a simple and intuitive way of distributing this TISM-key yet still allow for easy cross posting. To this end, a connector DAO was introduced, as shown in Figure 4.8. With this improved design, it is feasible to have an encrypted TISM (body), without having to re-encrypt a TISM for each cross-post. The connector DAO is inserted in front of the TISM pushing it down. The payload of the connector DAO holds the (symmetric) encryption key of the body of the TISM. This key, in turn, encrypted with the post-key of the target. By cascading the encryption, still only the read-key (of either target) is needed to decrypt the TISM. To cross post a TISM to some other target, a new connector DAO is created containing the TISM-key encrypted with the

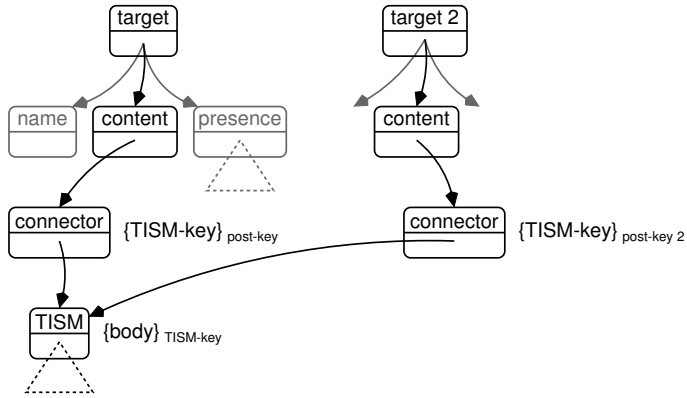

Figure 4.8: A Connector DAO is inserted in between a target's content DAO and a TISM DAO. The connector holds an encrypted version of the symmetric TISM-key. When a TISM is cross posted to target 2 a new connector DAO is created containing the TISM-key encrypted with the post-key of target 2.

post-key of the new target and the same TISM DAO is put in the cluster of this new connector DAO, see target 2 in Figure 4.8. Using a connector DAO means that only one key has to be encrypted and distributed. Without such a connector, a potentially large number of TISM parts would have to be retrieved, decrypted, re-encrypted, and distributed.

4.3 Spam

In their *"Email Statistics Report, 2012-2016"* the Radicati Group reports that in 2012 about 144.8 billion E-mail messages are sent daily. Around 90% of these message are spam or unsolicited E-mail, also referred to as, *abusive email*, *junk-mail*, UCE, or UBE. There is a simple reason why there is so much spam; *spam is effective*. The Radicati Group reports that despite adept antispam technology, still about 15% of all E-mail received is spam. Most of the cost of spam lies on the receiving side because that is where E-mail is stored and managed. The vast majority of spam recipients dislike it, as shown in Figure 4.9. However, low sending costs and a huge audience make spam cost effective, even if the response is a few per million.

Unsolicited messaging was first identified on USENET, and got its name from a Monty Python sketch. The first infamous professional spammer was Laurence Canter, co-author of the book *How To Make A Fortune On The Infor-*

Attitude about spam	Response
Like it a lot	1%
Like it somewhat	2%
Neutral	14%
Dislike it somewhat	20%
Dislike it a lot	63%

from: *ISPs and Spam: The Impact of Spam on Customer Retention and Acquisition.* Gartner Group. http://www.gartner.com/

Figure 4.9: What E-mail users think of spam.

mation Superhighway. Unsolicited messaging is a real problem, if not an outright threat to the sustainability of the Internet. The total worldwide cost associated with E-mail spam for 2009 is estimated at (than) € 90 billion (US $ 130 billion) [34].

4.3.1 Conventional Measures Against Spam

In several countries, antispam laws are in place or under consideration. It is doubtful whether any legislation will help. First of all, it will not help against *acquaintance spam;* spam following a solicited message. Second, the E-mail system is easily fooled, making it hard to track down the perpetrator. Also the international character of the Internet make it hard to enforce laws. There is a growing number of antispam products, which are offered by organizations like: Brightmail, MAPS, Postini, Trend Micro, Tumbleweed, ActiveState, Cloudmark and MailFrontier.

There are many antispam web sites (e.g., `http://spamcop.net/`). There are also a multitude of web sites (e.g., `http://www.byshenk.net/`) describing how to fight back against spam, how to complain to the spammer's ISP, how to create dummy web pages to poison *harvesters;* programs that collect E-mail addresses from web pages, how to prevent your machine from being used as an *open mail relay;* allowing spammers to use your computer to hide their identity, and so on. At the moment of writing, spam is largely vectored through E-mail, but spam has long since been a problem for the USENET system and it is also becoming an issue for wireless services and IM (such as AOL Instant Messenger, MSN

Messenger, YahooMessenger and ICQ). The latter type of spam is sometimes referred to as spim. Some computer-science professors even report to have gotten SMS spam in the U.S.

Message Filtering

Getting rid of unwanted messages, like viruses and spam, is hard to do. In principle, unwanted messages can be filtered out at any hop a message makes. For E-mail there are three logical moments to delete unwanted messages: at sendoff, in transit, and at reception. At each of these moments, messages can be *filtered* and automatically deleted. The filtering itself is simple: messages are run through an automated function that checks for known semi-unique patterns. If a message (including its header and attachments) pass some statistical threshold, it is deleted. For example, if a message contains the pattern *v1agra* chances are it is spam and the message can be deleted. The crux here is "*chances are*" for there is no certainty. However, it could, for example, be an important message from a manufacturer, warning about side effects. If so, it would be a *false positive:* a message falsely recognized as unwanted. Effective filtering will delete most spam messages (and almost all viruses) while occasionally mistaking a normal message for spam. Most people will not accept a filter that catches 92% of the unwanted messages at the cost of 2% false positives, however, 99.9% versus 0.01% would be accepted. There are many forms of filtering each with their own merits and demerits. Besides false positives there is one other demerit that all forms of filtering share: Filtering costs resources.

Adaptive-filtering or self-learning-filtering, has been proven to be very successful against spam. This type of statistical filtering takes place at the receiver's side taking corrections from the recipient. Besides statistic filtering, *blacklist filtering* is also popular. With this form filtering, a message is deleted if a from-field entry is known to be a spammer. Blacklisting has been effective against spam in the past, but spam has evolved to evade this filtering. Other filtering techniques against spam are *whitelisting* and *graylisting*. The whitelist filter allows only E-mail messages from known sources. If an E-mail message arrives from an unknown source, the sender is sent a challenge, and only if the other side sends the proper response, is the message accepted. Responding correctly, makes sending spam more costly but will usually be tolerable to the regular senders. The graylist filter rejects any E-mail message that has not been accepted recently, based on the triplet, (sender, receiver, sending-server). If the same E-mail is offered again after some time, typically 30 to 60 minutes, it is assumed that this message is sent by a full-blown SMTP server and it is accepted. Graylisting is (currently) effective because spammers use special programs to send their messages, and

these special programs try only once. One disadvantage of graylisting is the delay recipients experience when they get a message from a new sender. Usually a message passes several filters before it arrives at its end point.

Many users are unaware that their ISP has SMTP, POP, IMAP and Webmail servers that filter on spam and viruses. Of late, *spammers,* the people that send spam, have found new ways to fool filtering by means of inserting extra random words, by inserting images, etc. Basically filtering is part of an arms race between the spammers and the spam victims.

There are different, nonfiltering, attack vectors against spammers. First, sending messages should be made less cost effective, and second spammers should be traceable. The first vector is directed against the economics of spam, the second vector enhances the enforcing of antispam laws and a host of technical measures. These two attack vectors can be exercised when using a UMS as described in this section.

Attacking the Cost Effectiveness

There are several ways to make spamming less cost effective. If sending E-mail were to cost €0.10 per message, sending out billions of messages would instantly cease to be cost effective. However, even if the overhead cost of such a pay-per-message system turns out to be below €0.10 a message, people will never agree on what to do with the money. Some have suggested donating it to charity, some have suggested the recipient should receive it. However, all of these methods need strong protocols to make sure everybody complies. There are other ways besides pay-per-message, to increase the cost per spam message. Using *many* and *temporary targets* is one of them. One of the reasons E-mail spam is hard to eradicate is because it is expensive to invalidate an E-mail address. Once a spammer gets a hold of an E-mail address, he or she will be able to use it over and over again and he or she might also sell it to other spammers. So the cost of getting an E-mail address can be spread over many spam messages. When this happens, the user is faced with a difficult choice, abandoning his or her E-mail address, making it invalid for spammers and legitimate users alike, or dealing with the spam. This is the first place where the UMS system offers a better choice. It is very easy to create many targets, and when a target is dropped, the other targets will remain fully operational. Imagine a user setting up a target to communicate with his/her brother and one to communicate with his/her sister. Imagine a spammer somehow got hold of the post-key of the "brother-target." The user could *simply drop this tainted target*, create a new brother-target, and notify the brother that he should use this new target. The user will not even have to bother any other users about dropping the original

"brother-target." Since it is very cheap to make a target, it will even be feasible for any user to create a temporary target for any purpose. In short, a target is much cheaper than an E-mail box, or USENET group, hence the cost of getting a target has to be spread over only a few spam messages, increasing the cost per spam message.

The UMS has another feature that can be used to increase the cost overhead. In contrast to existing messaging systems, the UMS features a per-target replication. By selecting a *lazy* replication policy for public targets, the spammer has to provide for a cloud server that stays online, ready to deliver the spam every time the receiver requests it. A replication policy is said to be lazy if the data has to be kept at the origin. As discussed in Section 3.5.6 the MO system's basic replication is lazy. *Lazy replication makes sending spam more expensive* than the store-and-forward type of replication that is featured with most existing messaging systems, because the spammer will have to provide considerable long lived resources instead of being able to *fire and forget*.

Tracing spammers

With lazy replication, the cloud server functionality that a spammer has to provide will be very traceable. This opens an array of options of both technical and legal nature. Owners of cloud servers can be fined, for example, or a spam-hosting cloud server can be put on a blacklist. This option is not available with E-mail and USENET because of the store-and-forward replication and weak security model. Basically a spammer is hard to trace once the message has made the first hop. An ISP can easily see that someone sent a million emails (i.e., the first hop). But in many cases, the spammer owns the ISP.

Antispam Measures

The above mentioned effects on cost effectiveness of spam and possibility to trace spam back to the spammer, stem from the design of the UMS on top of the basic MO system. There are, however, specific measures that can be taken to make it even harder for spammers to operate. For example, an expensive posting replication policy could be implemented. Such a policy would necessitate a lot of computation before a message could be posted. One way of doing that would be to distribute only part of the post-key, forcing the posting party to calculate the missing parts by brute force. However, like with using a CAPTCHA (Completely Automated Public Turingtest to tell Computers and Humans Apart), some spammers will find a way to force the computer of innocent users to do the calculation. The proof-of-concept implementation does not feature any

such antispam measures.

4.4 Unique New Features

The proof-of-concept UMS shows some unique new features that one gets for free by integrating several messaging paradigms. For example, a user could receive an E-mail like message in a target, and they might follow up in an IM-like fashion if the sender is notably present in the receiving target. Another example would be that a message could originally been sent like an E-mail, and the receiving user might cross post it to a USENET-like group.

The UMS is also able to set up a replacement target for a target that has been compromised, for example, when a spammer has gotten a hold of the post and read-key of a target used for communication between two users, named, say, Alice and Bob. Alice would create a new temporary target, and this target plus its post-key are put in a TISM that is subsequently posted in the compromised target. Bob generates a new target and creates a TISM with a small message, the new target and both keys. This TISM is posted in the temporary target that Alice created. Alice will judge the small message to verify it is really Bob talking and both Alice and Bob drop the compromised target in favor for the new target that Bob made. With some additions, the UMS could even assist in identifying of Bob, for example, by deploying some signing algorithms. However, that was neither researched nor implemented. It is worth stressing that at no point in time did Alice and Bob lose contact, the "tainted" target was, merely, used to setup the replacement.

4.5 Storage versus Transport

Implementing the proof-of-concept UMS using the MO system, indicated that a full-blown UMS would be feasible. Since this proof-of-concept was also the first larger nontrivial program using the MO system, it also indicated that using the MO middleware layer did indeed simplify the design and implementation. Since the programmer-friendliness of the MO system is worthy of separate research, a quick qualitative observation follows below. From the experiment, it appeared that most aspects of using the MO system to build a distributed application, were either easy or easy explainable. However, one aspect appeared to be so subtle, that it was only encountered much later. The MO system can be viewed as a *storage and transport* system or as a *transport only* system, depending on the usage of expiration date and replication policy. If micro-objects are set to

expire after a reasonable delivery period, the micro-objects are used as a transport vehicle. If micro-objects are given a longer time to live and the application asks the cloud server to keep them alive even after that (by means of repeated setting of replication policy to extend the time to live), the micro-objects turn into storage objects. For a programmer unaware of this, it is easy to mix up these two ways of using micro-objects in the design phase. In this example, the TISM objects were designed as transport objects. The target objects (by nature) should have been implemented as longer lived objects. However, the design did not include the necessary prolongation of the *time to live* of the targets. The way the TISM tree was designed was contrary to the transport-only nature of the TISM objects them selves. These simple to fix design flaws were not detected during the testing phase, simply because there were no long-term tests performed. For mimicking the IM system, no changes are needed, however, for the E-mail and even more so for the USENET like functionality of the UMS some additions should have been made to the design from the beginning. In short, application programmers should be made aware of this matter before they attempt any serious development based on the MO middleware layer.

4.6 Application Programming Interface Design

The UMS API was modeled after the MO API. It uses the C language with a classical ADT design with rather verbose, underscore-based identifier naming. OOP (Object Oriented Programming) was considered but rejected because it would limit the number of systems the software could be easily ported to. The C language simply has the largest installed base, especially under relatively small systems. Name-space pollution is prevented by consequently prefixing identifiers with the proper ADT prefix. Classical pre and post fixing is used for user defined types (postfix: `_t`), structs (prefix: `s_`), unions (prefix: `u_`), etc. Every ADT is implemented as a pointer to struct. For example, `target_get_tism_list()` would operate on the target ADT named `target_t`, it would return a TISM ADT. In Figure 4.10 the main API of the UMS system is shown.

4.7 Future Improvements

A large part of the UMS layer has been implemented as part of a master thesis, the resulting system is reasonably functional and stable, however, more testing and more features would have been desirable. Below is a list of improvements

target	(target_)
TISM	(tism_)
body	(body_)
subject	(str_)
radix64	(str_)
text	(str_)

(a)

```
target_get_tism_list(tism_t **, target_t);
target_free_tism_list(tism_t **);
target_create(target_t *);
target_post_item(target_t *, tism_t);
target_get_radix64(str_t *, target_t, int flags);
target_radix64_to_target(target_t *, str_t);
tism_create(tism_t *, str_t, body_t);
tism_get_subject(str_t *, tism_t);
tism_get_body(body_t *, tism_t);
tism_get_radix64(str_t *, tism_t);
tism_radix64_to_tism(tism_t *, str_t);
body_put_string(body_t *, str_t);
body_get_string(str_t *, body_t);
```

(b)

Figure 4.10: (a) The main ADTs and their prefixes. (b) The main API functions.

that can be added to make the UMS more complete.

- As stated above, the design should be altered to issue *keep alive* messages for targets and TISM-trees.
- Currently the implementation is single threaded. The current MO system is thread safe. A real life messaging system probably needs a multithreaded implementation.
- Currently the UMS has a flat organization structure. By allowing targets to contain targets as well as TISMs, organizing large volumes of messages would become more intuitive to the user.
- Some additional tweaking of what data would go where could result in a gain of availability. For example, the body micro-object could contain the first body-part. Hence short messages and the start of long messages would be available two replication levels up.
- A full-featured presence system should be implemented featuring a range of possible states of presence like: away, busy, out for lunch, etc.
- Variable intervals could be used to lower the bandwidth needs for the presence messages.
- The current implementation allows a TISM to be part of one target only. Cross-posting should be added, for example, by using a connector DAO as shown in Figure 4.8.

- Currently the USENET newsgroup hierarchy is not mimicked, besides allowing targets to hold other targets, a special tree could be used to mimic it.
- The current implementation has been designed with protection in mind: a TISM should be protected with a unique symmetric key and this key should be protected by post-keys of the targets the TISM belongs to.
- The current MO system features dummy encryption, which should be replaced with industrial strength encryption.

4.8 Summary

Designing and implementing a proof-of-concept implementation has lead to some preliminary observations. First and foremost, *different messaging functionality can be unified* into one generic messaging system. Second, the MO middleware is relatively easy to understand and work with, it does seem to facilitate writing large-scale distributed applications. Third, one subtle concept needs explicit explanation: The differences between micro-objects as transport media and as storage media is not evident.

❧ *"How about performance and performance/resource trade-offs?"* That was the question. The answer was found in adding profiling hooks, designing a test application, adding replication strategies and repeated test runs. My promotor asked me the shortest question yet: *"So?"* Apparently (my) computer-science ideas are such that they need numbers and graphs to be convincing. *"What to measure how?"* I wondered. Performance overhead of the micro-object layer was one, a comparison for different replication policies was another. Running request/response tests in different address spaces but on the same computer would come close to testing the overhead of just the micro-object layer. Performance could be expressed as request/response times in milliseconds or application runtime in seconds. An interesting resource usage would be the total number of server requests. Running the same tests with different replication policies would show the effect replication has on the performance and the performance/resource trade-offs. Running the same tests using *mixed replication policies* would give still more graphs and numbers. Only after those tests would it be time to wonder: *"What are my conclusions?"*

Chapter 5

Micro-Object Experiments

> "A theory is something nobody believes, except the person who made it. An experiment is something everybody believes, except the person who made it."
>
> *Albert Einstein*, a man who thought time relative.

The micro-object middleware layer was designed to support large-scale messaging structures. However, the design has been kept generic and supports a broader range of large-scale distributed applications. The design offers a generic method to build arbitrary data graphs and a generic method to balance the trade-off between performance and resources. Chapter 3 describes how to construct those data graphs and how to set replication policies to balance the trade-off. In this chapter experimental data is presented to demonstrate some actual effects of the interaction between replication policies on the one hand and program data and logic design on the other. In particular it will be shown that:

> *Given a fixed data and logic design of an application based on micro-objects, changing replication policies can have a profound effect on the trade-offs of performance and resources.*

Besides a basic micro-object middleware layer, several straightforward applications were implemented. Most of these simulated some form of user-to-user messaging, but not all. For example, the UNIX (Uniplexed Information and Computing Service) domain sockets re-implementation allows some TCP-based applications to run atop the micro-object layer. This was not just an exer-

cise in stacking protocols, it allowed transparent traffic shaping and transparent end-to-end encryption, a bit like SSL does.

5.1 Implemented Replication Policies

The micro-object layer can accommodate many replication policies, three of which will be introduced in this section: BASE, FLOODING, and INTERVAL. Also a performance-measurement tool, called moping, will be introduced and several graphs of its output will be presented. In particular it will be shown that a change of replication policy has a profound effect on the trade-off of performance and resources. For all measurements, dummy encryption and dummy network communication were used to prevent delays caused by encryption or network latency from showing up in the output graphs. The cloud server and the lib-server have their own independent implementation for multilevel replication to lower the network load between cloud server and lib-server. The performance-measurement tool measures the resulting end-to-end latency.

5.1.1 Base Replication

Within the cloud server the BASE replication policy takes care of three things. First, it will store (encrypted) payloads. Second, it will share (encrypted) payloads upon request. Third, it will try to assent on cluster contents with peers and applications. The first two amount to basic pull replication, the third has been discussed at length in Section 3.7.1. Note, again, that this type of cluster synchronization is truly stateless and communication errors might interrupt a flow of ASSENT request exchanges, but eventually all clusters will synchronize. On the application side, all newly created micro-objects will be rapidly forwarded by the lib-server to the cloud server that acts as the home server. In contrast, cluster additions are not instantly forwarded by the lib-server. This might seem counterintuitive. However, not forwarding allows other replication policies maximum freedom to do as they see fit. Currently all (i.e., both) non-BASE replication policies instantly forward cluster additions. The sole reason fresh micro-objects are always instantly forwarded, is that tokens of micro-objects sometimes need to be forwarded out-of-band, not so with clusters. The BASE replication is implemented and tested in full, exactly as described.

5.1.2 Flooding Replication

Within the cloud server the FLOODING replication policy honors its name by forwarding any addition to a micro-object's cluster, to all remote parties that have requested or inherited this policy for that micro-object. The current implementation keeps a ledger of token/remote-address pairs. For each pair a (minimal) content list is kept that is assumed to represent the remote copy of the cluster. When a cluster change needs to be forwarded, an ASSENT request is faked using that minimal content list. This means that the system is confronted with a (seemingly remote) BASE replication ASSENT request. Put in another way, the FLOODING replication acts as a proxy to request an update of a cluster when it has been changed.

Note that no actual state information is kept in sync between the parties. The cloud server simply assumes perfect communication in keeping track of what the remote site "knows." This is a reasonable assumption, since most messages make it across. However, thanks to the BASE replication, an incidental miss will be corrected. If the cloud server assumes wrongly that some tokens are known at the remote side, the result is a few extra ASSENT requests being sent back and forth. If the cloud server wrongfully assumes some tokens are not known remotely, the BASE replication will be forced to send too much information, which will be discarded by the remote side. To sum up, the FLOODING replication policy fakes a reasonable number of BASE ASSENT requests on behalf of remote applications as soon as a token is added to a relevant cluster. On the application side, the lib-server forwards any cluster additions instantly to the cloud server. This allows the cloud server to further cascade the addition. Since there is currently no encryption of cluster additions, the lib-server does not need to instruct the cloud server what to do. When this encryption is implemented, the application side replication will have to be extended.

5.1.3 Interval Replication

Within the cloud server the INTERVAL replication policy is like the FLOODING policy, with one major difference. Changes are not flooded immediately but at fixed intervals. If several changes happen in one interval, the changes are sent out together, trading WAN (Wide Area Network) bandwidth for latency. The current implementation uses one dirty flag per token/remote-address pair. When a cluster addition is detected, the relevant dirty flags are raised but no ASSENT requests are injected into the system, yet. At fixed intervals the flags are checked. For every raised dirty flag the corresponding remote address is sent an ASSENT request and the dirty flag is cleared. In short, the cloud server side

INTERVAL replication policy records all cluster additions and sends out groups of cluster additions at fixed intervals. The purpose of INTERVAL replication is to save WAN bandwidth and application to server communication predominately uses LAN (Local Area Network)s. Therefore, on the application side, the INTERVAL replication acts exactly like the FLOODING replication policy, forwarding cluster additions instantly.

5.2 Implemented Security Policies

As with replication, payload security is provided through policies. The current implementation does not contain proper security policies. However, weak forms of symmetric and asymmetric encryption for payloads were implemented. This was necessary to test the API and to get more realistic time measurements. Strong encryption policies should be added.

Also the current implementation uses MD5 (Message Digest algorithm 5) as an one-way hash function (see [52]). Wang and Yu have shown that MD5 is not collision resistant [68], a better hash algorithm may need to be used. The current implementation allows for the usage of multiple hash functions and adding or replacing MD5 will not affect the rest of the system.

The cluster security infrastructure has been implemented, but currently no cluster security policies have been implemented. Those should be added both at the cloud server level as well as at the lib-server level. At the cloud server level, authentication policies should be added to guard the addition to the clusters. Even if payload security policies would make it impossible to add a valid micro-object to a cluster, authentication allows cloud servers to weed out invalid additions early on. At the lib-server level, policies for nonrepudiation and end-to-end encryption should be added for applications needing these types of policies. Encrypting cluster additions end-to-end will hamper the replication at the cloud server level, necessitating application participation in the replication process.

The current implementation also does not hinder traffic analysis or resist DoS attacks. Traffic analysis can be hampered by the application independently from the micro-object layer. Also traffic analysis can be made more difficult by replication policies. A DoS attack can be more effectively countered by the lower-level network layers. However, a flash crowd might have a similar effect as a DoS attack. Therefore a complete implementation would have some mechanism to redirect (a set of) requests to another cloud server. Typically, a cloud server should keep track of how often data are requested and who requested them. If some information becomes too popular, a cloud server could redirect

a request and all its related requests to an cloud server that recently requested the same information. This, however, is not currently implemented.

In short, security is a big part of the design but the current implementation is lacking most of it. Doing security well is not a trivial matter, best left to a seasoned expert. The implementation, however, is fully prepared to host the necessary security code. One could argue that security not done is better than security not done well.

5.3 Moping

One of the simplest tools that was implemented is moping (pronounce as two words: *mo-ping*). It provides a similar service as the ping tool for measuring Internet routing latency (see [38]). Moping measures the time it takes to "send" information from a client application to a server application and back, even though the micro-object middleware does not provide a "send" functionality. More precisely, two moping programs, one running as a client and one as server,

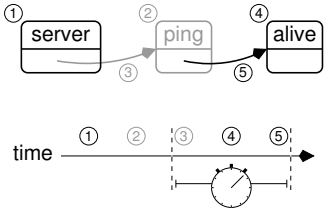

Figure 5.1: The moping server creates a server micro-object ①, the moping client creates ② and clusters ③ a ping micro-object to that, to which the moping server creates ④ and clusters ⑤ an alive micro-object.

together construct a data graph in a fashion that can be interpreted as sending and receiving message objects. Stated another way, the moping application can measure how long it takes for graph changes to propagate between applications. The exact graph is shown in Figure 5.1.

The moping application is started in server mode. It creates a server micro-object with `mo_create_new()`. Then, it hooks a callback function onto the server micro-object using `mo_put_cter_clbk()`. It continues with creating a string, encoded in radix-64 [15], containing the token and necessary security details, utilizing `bfer_encode_radix64()`. After that, the main thread goes to sleep, leaving it to the callback function to respond to the ping micro-objects with alive micro-objects. Another moping application is started in client mode,

possibly on the other side of the Internet. When the moping client is given the string that was generated by the moping server earlier, it creates a local copy of the server micro-object, by calling `mo_create_copy()`. It then creates a brand new ping micro-object, starts a timer and puts the ping micro-object in the cluster of the server micro-object, by means of `mo_cter_add_mo()`. The moping client application then blocks on a `mo_cter_wait()` call, until an addition to the cluster of its ping token has been received. It stops the timer and displays the number of milliseconds it took. Then it repeats these steps from creating a new ping micro-object till displaying the time, over and over again. Figure 5.2 shows the resulting application object, as a graph of micro-objects, after several rounds of adding ping and alive micro-objects.

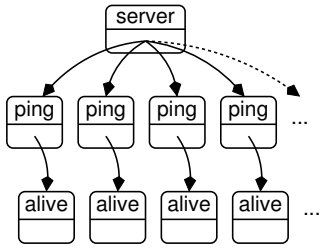

Figure 5.2: The moping application object after several rounds of propagating ping and alive micro-objects.

5.3.1 Moping Testing Platform

The micro-object middleware layer was developed (and runs) on many platforms, including a vanilla PC running Microsoft Windows XP, a G4 iMac running Apple OS X, a vanilla PC running FreeBSD, a big AMD-64 multiprocessor running Linux, a Sun Sparc running SunOS, and several Intel-based Apple Mac's. The timing experiments were all performed on a DELL Inspiron Mini 9 running Mac OS X 10.6.3 on an Intel Atom at 1.6 GHz, because that was by far the slowest machine available at the time. Doing timing on any other machine resulted in flat results, over ninety percent of the outcomes were either 1 ms, 2 ms or way above 1000 ms. With the one order of magnitude slower DELL, timing showed more variation. This is much more suitable to do a relative comparison of the outcomes of the experiments. All the processes were run on the same computer. This was done to neutralize the substantial overhead that would be caused by the underlying physical network. The properties of the physical network lie outside the focus of this chapter. As a by-product, some

outcomes can be used to estimate an upper limit of the overhead induced by the middleware layer as a whole. Since the current implementation is in no way optimized for speed such an upper limit should not be taken too seriously. On the other hand it is encouraging to note that the fastest moping results, even on this slow platform, lay well below the 10 ms.

5.3.2 Moping with Base Replication

Clearly the moping client is in for a long wait if the BASE replication policy is used exclusively. One might wonder why BASE replication does not implement some form of forwarding cluster additions. The reason is simple: No information should be pushed at an application unless the application explicitly asked for it by setting a replication policy with `mo_put_repl()`. Stated more precisely, no diffusion of information should consume resources other than those provided by the source and all willing recipients: the source and willing recipients, in order to receive the information, will have to participate in a replication-specific overlay network.

5.3.3 Moping with Flooding Replication

Adding the FLOODING replication policy to all the micro-objects will enable forwarding of cluster additions, as shown in Figure 5.3. Setting the replication

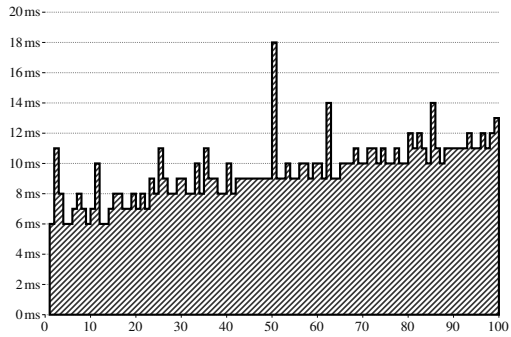

Figure 5.3: Output of moping with FLOODING replication. The first 100 iterations on the x-axis are plotted against response time in milliseconds on the y-axis. Note the upward trend.

policy of just the root node (i.e., the server micro-object) will suffice, if the appropriate replication level is used. So both the moping server and the moping

client use `mo_put_repl()` on the server micro-object: the moping server right after creating it with `mo_create_new()`, the moping client right after creating a local copy with `mo_create_copy()`. Both will use a replication level of 4 (or more) to assure that (1) the server cluster's changes, (2) the ping payload, (3) the ping cluster's changes, and (4) the alive payload are flooded. Note that from the description above, a level of 3 would suffice, because there was no mention of payloads. However, the current implementation of moping puts an auto-incremented number in the payload of every ping micro-object and this number (with 1000 added) is put in the payload of the alive micro-object. The current implementation prints out two intervals, one after receiving the token of the alive micro-object and one after receiving the payload. For this chapter, all the interval durations include the retrieval of the payload of the alive micro-object. The numbers were very close, in almost all cases the difference was 1 ms. Plotting both would not make for a very interesting graph.

5.3.4 Moping with Interval Replication

Figure 5.4 demonstrates the output of the moping exchange with INTERVAL replication. The graph looks quite dull. Other than the first iteration, all round

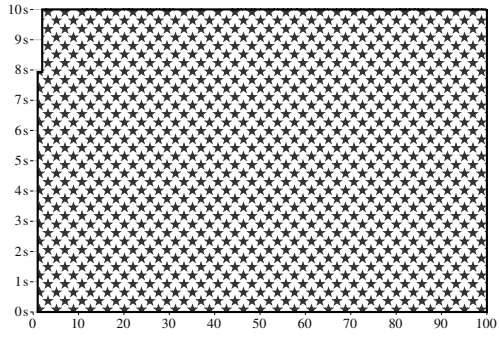

Figure 5.4: Output of moping with INTERVAL replication. The first 100 iterations on the x-axis are plotted against response time in *whole seconds* on the y-axis. *Note that the first iteration is about two seconds shorter.*

trip times take about 10000 ms or 10 seconds. This dullness is intriguing: Why are the values so similar? The current implementation of the INTERVAL replication policy uses an interval of five seconds between starting to check the dirty flags. Since two additions have to be forwarded, a maximum of just over 10

seconds is to be expected. Why does this implementation behave so poorly and maximizes the delay? And why does only the first iteration take 2 seconds less than the rest? The latter question is easy: the testing script puts 2 seconds between starting the cloud server and starting the moping client. Clearly some synchronization takes place during the first iteration. The synchronization is caused by the fact that 5 seconds is a very long time compared to the time other actions take. This means that as soon as a change is forwarded, a response is returned nearly instantly. This response will always have to wait just under the full interval time before it is forwarded. Figure 5.5 sketches what is going on. Actually, 10 seconds is an abnormally short time, normally there are two cloud

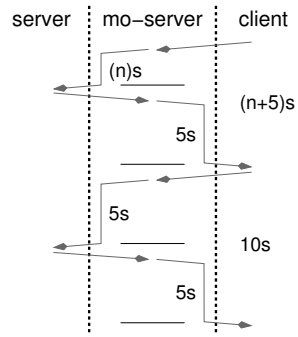

Figure 5.5: After half a round trip, forwarding is locked in five second intervals. With two delayed steps, the round trip time is 10 seconds. Note that $n \leq 5$.

servers involved. So normally a round trip would take 15 seconds, as shown in Figure 5.6. To improve the INTERVAL replication policy, one might consider to alternate a short and a long interval. This would cause instant responses to be mostly forwarded after the short interval.

5.3.5 Moping with Mixed Replication

Note that some replication policies can be effectively combined. Figure 5.7 shows the results of just such a combination. The moping server was using the INTERVAL replication and the moping client was using the FLOODING replication. This means that everything will be flooded to the moping client where everything for the moping server is sent off at an interval. The rationale behind this choice is that it would make sense to group changes to the server micro-object, for its cluster is ever growing. For the individual ping micro-objects, there is no reason to wait for other additions that can be lumped into one ASSENT request,

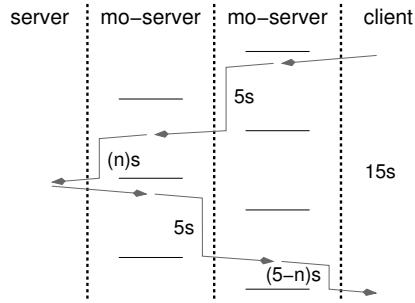

Figure 5.6: With two cloud servers involved and forwarding locked in five second intervals will cause a delay of 5+(n)+5+(5-n) = 15 seconds. Note that $n \leq 5$.

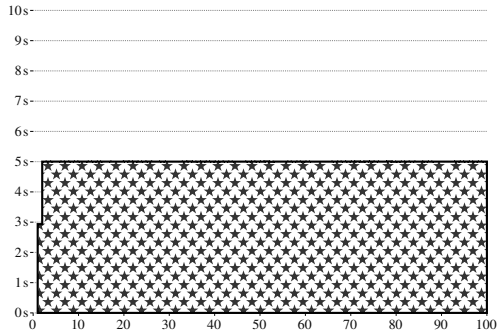

Figure 5.7: Output of moping with a mix of INTERVAL and FLOODING replciation. The first 100 iterations one the x-axis are plotted against the response time in seconds on the y-axis. Note the similarity of this graph and the one in Figure 5.4.

for there will be at most one cluster member, as shown in Figure 5.2. As can be clearly seen in Figure 5.7 not waiting for 5 seconds when forwarding the additions to the ping clusters, shaves off 5 seconds of every iteration. The rationale behind this choice, however, is false. The moping client is synchronous, i.e., the moping client waits for the outstanding alive micro-object before a new iteration is started. And as long as the moping client is synchronous, additions of ping micro-objects to the cluster of the server micro-object cannot be transferred in groups. To see the full effect of the INTERVAL replication, an asynchronous version of moping is needed.

5.4 Asynchronous Moping

An asynchronous version of moping differs only a little from the synchronous one, in the current implementation, at least. Since callback functions in the current implementation are re-entered, the moping server operates fully asynchronously. In the code of the moping client, the `mo_cter_wait()` call and subsequent handling of the timer has to be replaced by a callback function that handles the timer and a call to `mo_put_cter_clbk()`. In order to control the load on the system, the moping client process does wait for a number of milliseconds in between sending a ping micro-object. In practice a value of 100 ms seems reasonable. This duration makes sense, since it is much longer than the minimum time a round trip takes. There should be hardly any difference for the flooding example, for the moping client will actually be waiting longer than when it would wait for the alive response.

5.4.1 Asynchronous Moping with Flooding Replication

Figure 5.8 shows the asynchronous version of moping in action. It looks a lot

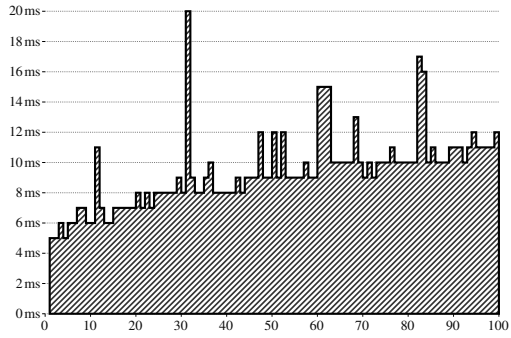

Figure 5.8: Output of the asynchronous version of moping with FLOODING replication. The first 100 iterations on the x-axis are plotted against the response time in milliseconds on the y-axis. Note the similitude of this graph and the one in Figure 5.3

like Figure 5.3. The total running time is well over 10s (12.048 ms) as is to be expected of a program that waits 100 times for 100 ms. The total running time of the synchronous version was just over 3 seconds (3.179 ms). Switching from the synchronous to the asynchronous version did not do much for the average

round trip time when the FLOODING replication policy is used.

5.4.2 Asynchronous Moping with Interval Replication

Switching to the asynchronous version drastically impacts the outcome when the INTERVAL replication policy is selected. The asynchronous output in Figure 5.9 looks rather different from the synchronous output of Figure 5.4. The

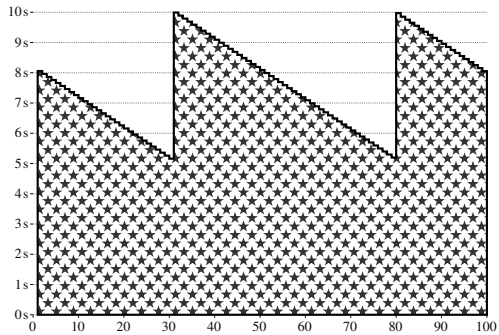

Figure 5.9: Output of the asynchronous version of moping with INTERVAL replication. The first 100 iterations on the x-axis are plotted against the response time in seconds on the y-axis. Note the drastic change in shape compared to the synchronous version as shown in Figure 5.4

INTERVAL replication policy saves up the first three seconds of cluster additions before they are sent in bulk from the cloud server to the moping server process. With a 100 ms wait in between pings that amounts to about 30 packets. The first packet has to wait for about 3 seconds the last makes its way instantly to the moping server process. Then the next 50 pings are lumped together in one ASSENT request and then the remaining 20 pings are sent over (after waiting between 5 to 3 seconds). Up till this point, there has been no reason to write about the trade-off between latency and resource usage. No matter what combination of FLOODING and INTERVAL replication was used, if the communication was synchronous, 100 ASSENT requests were sent from the cloud server to the moping server process: one for every addition to the server micro-object's cluster, hence 100 ASSENT requests. The asynchronous version of flooding will likewise use 100 ASSENT requests. With asynchronous moping and INTERVAL replication, however, that number is reduced from 100 to 3. So here is the trade-off: asynchronous moping with FLOODING replication costs 100 ASSENT

requests and delivers an average ping time of 9 ms. Asynchronous moping with INTERVAL replication costs 3 ASSENT requests and delivers an average ping time of 7.583 ms. So in this particular case INTERVAL replication is over 800 times slower, but at 3% of the communication costs. Note that there are other considerations when selecting a replication strategy. For example, the huge difference in variance might be important, or the total running time.

While running tests, it was interesting to note that there was actually one test where INTERVAL replication outperformed FLOODING. When the time between pings was lowered to 1 ms, and a slow computer was used, the FLOODING replication started trashing the system around the 200th packet. Soon, round trip times of over 20 seconds were no exception. The system also did not recover from this state and gradually grinded to a halt. Running the same test using the INTERVAL replication had no noticeable negative effect. The replication policy had no problem at all to cope with the torrent.

5.4.3 Asynchronous Moping with Mixed Replication

As noted in Section 5.3.5, there is a rational for using a mix of replication policies with moping. Since the moping server process waits for 100 updates to 1 cluster and the moping client process waits for 1 update to 100 clusters, there is no use for the moping client process to try and bundle updates. A five seconds shave is to be expected for a run with mixed replications. Sure enough, as shown in Figure 5.10, the output resembles the previous output with 5 seconds off the duration of every iteration. Asynchronous moping with FLOODING replication is thus seen to cost 100 ASSENT requests and delivers an average ping time of 9 ms. Asynchronous moping with a mix of FLOODING and INTERVAL replication costs 3 ASSENT requests and delivers an average ping time of 2.560 ms. So in this particular case using this replication mix is over 280 times slower, but at 3% of the communication costs.

5.4.4 Summary

The current implementation of the micro-object middleware layer replication policies has shown the first signs of life. The first tests are encouraging, showing that even these relatively primitive policies make it easy to influence the (emerging) behavior in terms of trade off between latency and resource usage. An overview of the relevant numbers mentioned in the previous sections are given in Figure 5.11.

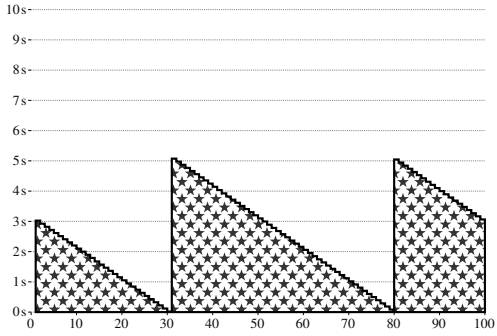

Figure 5.10: Output of the asynchronous version of moping with a mix of INTERVAL and FLOODING replication. The first 100 iterations on the x-axis are plotted against the response time in seconds on the y-axis. Note the similitude of this graph and the one in Figure 5.9

100 Ping requests	Server policy	Client policy	Assents to server	Response time	Client runtime
Synchronous	FLOODING	FLOODING	100	9 ms	1.05 s
	INTERVAL	INTERVAL	100	9.982 ms	998.84 s
	INTERVAL	FLOODING	100	4.980 ms	498.82 s
Asynchronous @ 100 ms intervals	FLOODING	FLOODING	100	9 ms	10.81 s
	INTERVAL	INTERVAL	3	7.583 ms	18.85 s
	INTERVAL	FLOODING	3	2.560 ms	13.82 s

Figure 5.11: Trade-off between the number of assent requests a ping server receives and the average response time when clustering 100 ping objects to a server object.

❧ *"What are my conclusions?"* The final question at last. Getting abstractions right helps me understand stuff better. Truly understanding a nontrivial matter makes me euphoric. If only I understood why people keep using fundamentally flawed models … *"Any scientific conclusions?"* my promotor interrupted my self-psychoanalysis. *"Ah, those,"* I responded. Clearly there was work related to unified messaging and also related to distributed-programming frameworks. As to further research into both unified messaging and micro-objects, we have only just begun our journy. I wonder: *"Who will join us on this journy?"*

Chapter 6

Conclusions

> "We are not certain, we are never certain. If we were we could reach some conclusions, and we could, at last, make others take us seriously."
>
> *Albert Camus*, French-Algerian writer and goalkeeper.

As it turned out, this thesis has two main topics: *unified messaging* and *micro-objects*. Both subjects are about communication. The micro-object layer is about machine-to-machine communication and unified messaging is about user-to-user communication, the distinction warrants separate concluding statements.

6.1 Work Related to Unified Messaging

The term "unified messaging" is used in a number of commercial and research areas with varying connotations. There is no single, broadly agreed upon, formal definition of "unified messaging" it can refer to simple forwarding of faxes and voicemail messages via E-mail, or to sophisticated, format changing, multicopy delivery. Maybe because of this, "unified messaging" held the number one position on the 1998 Wired Magazine Hype List.

Currently Google can find over ten million related pages for the term "unified messaging." Google (`scholar.google.com`) currently lists over six thousand related papers and patents. The top ten of these are all patents registered by the USPTO. Without exception, the description of these patents are about how to

connect current systems to integrate a heterogeneous set of messaging systems by connecting them through gateways. There are only a few nonpatent papers in the top twenty, the first [7] is about solving personalization on top of integrated messaging services, the second [57] is about getting voicemail from the telephone system in to the E-mail system, and the third [20] is a description of more generic way, using gateways, to forward messages over several existing systems.

This is in line with the observation that a lot of interest in "unified messaging" is commercially motivated and usually provides a centralized storage connected to different systems through gateways, services to deal with differences in data representation, and multiple ways to access stored messages. An example would be Nortel's CallPilot Unified Messaging. It collects voicemail, fax and E-mail on a single location that can be accessed both via the Internet and by telephone. Most of these commercial "unified messaging" product fall in the category of CTI (Computer-Telephony Integration).

The ACM Digital Library holds forty papers related to unified messaging. Some publications are about unrelated things like interprocess messaging [1] and there are several that only mention the term once. One paper [29] states "unified messaging, which has been 'almost here' for too many years to count," without further elaboration. The ones that are elaborating on unified messaging, all are about message routing, through existing systems, to mobile users or handling different representation of message content. For example, Barber [10] claims "Unification is achieved by reducing all incoming messages to a messaging-independent format and placed in a single message store. This means that any incoming message can be piped onwards through any other compatible means of communication." None of the ACM publications are about *replacing* messaging systems by one unified system.

This thesis does not focus on common denominator message format *reduction*, because it does not focus on central storage and delivery through existing systems. It does focus on *replacing* current messaging systems by one system that can mimic the behavior of the systems it replaces, independent of the format of the message. Hence, all above mentioned publications describe marginally, to nonrelated research, with the exception of the one mentioned below.

There are a few related papers describing research that relates to the unified messaging system described in this thesis. Like the paper on CLUES [45], that describes a system that uses machine learning techniques to determine levels of notification that in turn can be used to determine the appropriate form of message delivery and communication. If proven effective in determining the right notification level, it could prove valuable for selecting a specific behav-

ior, per message, of the unified messaging system. For the same reason the aforementioned Universal Inbox project [7] at UC Berkeley, presenting an architecture that enables redirection of incoming communication based on user preference profiles, might have a meaningful implementation on top of the system as described in this thesis. Another such related research is the Mobile People Architecture project at Stanford University, which views "the person" as the message end point (not a machine) to maximize user reachability [6]. Like this thesis it focuses explicitly on user-to-user (they call it person-to-person) messaging. However, this research suggests to replace end-point delivery with proxy delivery, unlike the approach in this thesis, where end-point delivery is but one of the delivery options.

Healy, Barber, Nolan [30] explicitly list the main components of their unified messaging system, as shown in the left-hand column of Figure 6.1. The right-hand column of Figure 6.1 lists the corresponding main components of the unified messaging system as described in this thesis.

Healy, Barber, Nolan UMS	**Target / TISM based UMS**
A single inbox per user for all messages.	Many targets per user.
Media conversion (e.g. text-to-speech).	No conversions.
A unified message store.	A distributed message store.
Unified management of disparate messaging components.	A unification of disparate messaging components.
A unified command set.	A unified API.
Unified directory services.	No directory service.
Rules-based forwarding of messages.	Emergent behavior forwarding of messages.

Figure 6.1: A comparison between Healy, Barber, Nolan and Target / TISM based unified messaging.

Another difference between all the above mentioned research and this thesis is that none of the research is overly preoccupied with either security or scalability. There exists some research on user-to-user messaging that does focus on these two important aspects. For example, Wrzesińska [72] shows "that it is possible to design and implement an instant messaging system [...] [that] satisfies the defined security and scalability requirements." This research, however, focuses on just one type of user-to-user messaging (i.e., instant messaging) rather than on generic user-to-user messaging.

6.2 Work Related to Micro-Objects

The term *micro-object* was derived from two other well known computer science terms. *micro kernel* and *distributed object*. *Micro* for simplicity and small size and *object* for application building block. Since building blocks, in a distributed system, have to be exchanged and stored, the MO system could also be viewed as a data exchange and data cloud store system. This view is taken in Section 6.2.2.

6.2.1 Micro-Objects as Building Blocks

When micro-objects are viewed as small distributed building blocks for larger distributed-application objects in order to facilitate large-scale distributed computing, its goal resembles the goals of middleware layers, like CORBA and EJB. Both try to hide certain aspects of the distribution of data over several physical locations.

CORBA

The CORBA middleware layer has a classical RPC model, most suited for OOP programming languages with exception handling, like JAVA. Figure 6.2 shows CORBA's basic communication stack. The CORBA middleware layer has grown over the years and CORBA 3.x has many features that facilitate writing a distributed application. For example, it has a generic way of describing an object's interface to the system (i.e., an IDL). The current version of CORBA tries to make transfer failure as transparent as possible to the application layer. When confronted with a transfer failure that cannot be resolved automatically, CORBA generates a detailed, platform-specific *exception*. It is up to the application layer to handle the those exceptions. Remote state changing RPCs, i.e., `ojb.a<-set(1)`, will suffer from the infamous *last* ACK problem independent of the protocol design used [4]. Chances of failure, however small, will limit the scalability of any system.

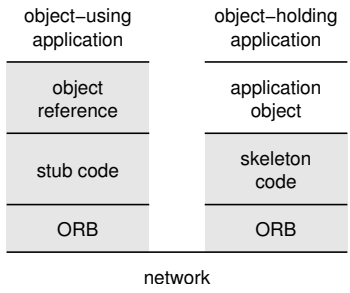

Figure 6.2: The base CORBA architecture. Only the application object and the applications using and holding it, have to be provided, all the other layers (in gray) are either generic or can be generated by the system with minimal assistance.

Enterprise Java Beans

EJB provide three important types of client/server communication services: session tracking, access to entities (usually a database) and message exchange. Most of the principal workings of EJB are like CORBA's. Actually, EJB 1.x used RMI (Remote Method Invocation) built on CORBA's RPC. After about a decade of development, EJB 3.x, though vastly different from the 1.x version. Its core still offers distributed computing by remote execution, however. In EJB 3.x there are three types of distributed objects (called beans), two use RMI: *stateless beans* and *stateful beans*, as shown in Figure 6.3(a), and one uses asynchronous messaging: *message-driven beans*, as shown in Figure 6.3(b). Actually, the standard does define a fourth remote object type, a variant on the stateful type, named *entity bean* offering database-like features. All remote objects are grouped on servers in so called *containers*. The stateless objects can be used by many clients, the stateful objects are bound to a single client process, and only the message driven objects are capable to receive state changing instructions from multiple clients. The programmer has to implement an application object class. On top of that, a programmer also has to specify some meta data called a deployment descriptor. The latter allows the development system to generate the support code.

CORBA and EJB versus Micro-Objects

In short, CORBA and EJB differ both from each other as well as from the MO system. The biggest difference between CORBA and EJB on the one side and the MO system on the other, is that the MO system always delivers the data

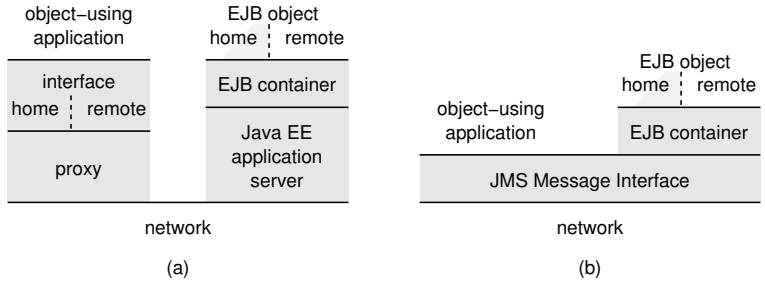

Figure 6.3: The base EJB architecture. (a) Simplified layout for stateless and stateful remote objects and (b) for message-driven remote objects. Only the EJB object, its remote interface, part of its home interface and the applications using it, have to be provided by the programmer, all the other layers (in gray) are either generic or can be generated by the system with minimal assistance. The remote interface implements the RMI calls, the home interface implements additional local methods.

to the processing units, where the other two send processing commands to the server holding the remote object. Also the MO system has very few processing operators, compared to the other two systems. This is because the distributed-computing paradigms of both EJB and CORBA try to transform application objects into distributed-application objects, forcing both systems to send data-manipulating messages to remote data with various degrees of communication transparency. The micro-object system takes the reverse position, where data travels towards the instructions and there is no communication transparency. Both EJB and CORBA differ in many other ways from the micro-object system but most of these differences are less fundamental than the differences mentioned above and could probably be remedied with additional coding. One such difference would be that the MO system is designed to have multiple replication policies active at the same time for a given single object. Another would be that the micro-object system features a generic way to connect application objects. There is, however, another cardinal difference between traditional distributed computing and the MO system. Where traditional systems need to know and use the network address of the peer or server that holds (a copy of) the data needed, the MO system addresses the data directly. There are two related distributed computing paradigms that use a comparable notion of addressing data: *content-based networking* and *content-centric networking*. With content-based networking, data is retrieved for processing using *predicates*, with

content-centric networking, data is retrieved by various forms of determination. The MO system shares the addressing model of these two only with respect to the fact that the data is addressed directly and not indirectly through network end points. However, the MO system offers a generic and simple way for the creators of data to connect it into one (or many) graphs, and the consumers can either walk the graphs, or select to be notified when certain parts change. Below a more specific comparison.

Content-Based Networking

As Carzaniga et al. write [16] about the extreme case where no address is used other than a reference to the content. "Under such an approach, producers will generate messages, but with no particular destinations intended. The destinations are determined by consumers expressing interest in the delivery of messages satisfying some arbitrary predicates on the content, independent of the producers of the messages." Using such an "extreme" content-based communication service, receivers declare their interests and senders publish content. The routing of data between them becomes solely the task of the supporting middleware. There are three major variants of data exchange. First, a *channel-based* variant, where receivers and senders listen to and talk to communication pipes. Receivers select pipes of interest and maybe even filter out some data for further fine tuning. Second, a *subject-based* variant, where senders add a subject to each unit of data. Receivers use predicates to express what subjects they are interested in. Third, a *content-based* variant, where receivers use predicates for arbitrary data selection across the entire content of data units.

Besides the three principal ways of data addressing (or "event-notification"), *pipe-based addressing*, *subject-based addressing*, and *content-based addressing*, the MO system uses a fourth, more primitive form: *token-based addressing*. With the first three forms data objects are requested by specifying criteria or predicates. With token-based addressing a data object is requested by specifying the unique token of the exact individual content. With pipe-, subject-, and content-based addressing, data objects can be grouped implicitly by their address. With token- and pipe-based addressing, grouping data objects has to be done explicitly. With subject-based addressing, grouping can be done explicitly by specifying a unique group id as part of the subject, or implicitly by not doing so. Content-based addressing does not allow such explicit grouping, because it would mean changing the content. Different addressing results in different data flows. For example, a token can be used to pull data from the system, where data is pushed with pipe-based addressing and both subject- and content-based addressing lead to a rendezvous type of predicate data and data

objects. Figure 6.4 shows a comparison of these four types of data addressing.

	Token	**Pipe**	**Subject**	**Content**
Address provided by producer	Unique id	Group id	Free form	Implicit
Address provided by consumer	Unique id	Group id	Predicate	Predicate
Data object grouping	Explicit	Explicit	Ex-/implicit	Implicit
Data flow	Pull	Push	Rendezvous	Rendezvous

Figure 6.4: A comparison of token-based, pipe-based, subject-based, and content-based data addressing.

Content-Centric Networking

The micro-object system also shares the *name-data binding* paradigm with the proposals championed by Van Jacobson [32]. Compared to the micro-object system, these proposals feature named content utilizing a far more complex form of addressing, involving *augmenting names with time/version* and *nicknames*. Also, Jacobson's data distribution is based on *dissemination* (unacknowledged data push), where micro-object distribution is based on *unacknowledged data pull* with optional replication allowing data push as well as acknowledged data transfer. Like the micro-object system, Jacobson's system uses end-to-end encryption. Jacobson's design is *context-aware* which enables named content to migrate wherever it is needed. The data flow is driven by predicate-based requests and the content of the messages. In contrast, the micro-object layer is totally content agnostic, it offers only explicit connection between data objects. The data flow is driven by explicit requests, explicit data connections, and explicit replication policies. Currently, CCNx[1], a first incarnation of Jacobson's system, is available as an open source project. It trades *named hosts* for *named content.* Like CORBA and EJB, Jacobson's proposals are about augmenting relatively large objects, in stark contrast to the micro-object system, which is about building relations between relatively small immutable objects.

[1]According to the CCNx homepage, http://www.ccnx.org/, *"CCNx® is an open source project in early stage development exploring the next step in networking, based on one fundamental architectural change: replacing named hosts with named content as the primary abstraction."*

Publish-Subscribe Messaging

Sharing blog or news entries is very popular on the Internet. Referred to as RSS or Atom feed, information is fed to a (potentially large) number of subscribers. It is similar to USENET, though with a different namespace. As shown in Figure 2.3 on page 51 it would be quite easy to mimic using unified-messaging.

6.2.2 A Cloud Of Micro-Objects

Forgoing underlying paradigms, from a more abstract point of view, the micro-object system could be viewed as a data messaging system as well as a data cloud store. With some caveats, the latter view can be taken to compare the micro-object system to other data cloud systems. Though it is difficult to arrive at a canonical definition, let us use the following high-level definition for *data cloud:* a collection of data-object stores acting as one, featuring fault tolerance, scalability, data versioning, access control, and eventual data consistency. With the exception of versioning this is in line with what the micro-object system was designed for. Versioning can be implemented by creating new objects for every version and a suitable graph to connect the related objects as we will demonstrate below.

Comparing a micro-object based system to a data cloud system is inherently unfair since cloud storage is most often used in conjunction with *cloud computing*, bringing data processing close (in terms of latency and throughput) to the data cloud. For the micro-object system to be comparative, it has to be augmented, allowing user applications to run close to the home servers and enabling fast access to the local micro-object store. Such an augmented micro-object system could be compared to cloud stores like *Amazon S3* and *Google Cloud Storage*.

Both Amazon S3 and Google Cloud Storage provide large (tera-byte sized) data objects and meta data, grouped into buckets, addressed by keys, and protected by access control lists. Individual objects can be addressed using URLs like: `<http://store.googleapis.com/bucket/key>` (Google Cloud Storage) and `<http://s3.amazonaws.com/bucket/key>` (Amazon S3). By comparison, micro-objects are limited. The payload is in the microscopic kilobyte range and the meta data is restricted to other object's addresses (tokens). Also Amazon S3 and Google Cloud Storage feature more flexible access control lists than the current implementations of payload and cluster security. Grouping by bucket is relatively coarse in comparison to the micro-object grouping that allows arbitrary graphs. These differences dwindle when the logical layout of aforementioned cloud storage systems is mimicked by building a `cloud` DAO as depicted in Figure

6.5.

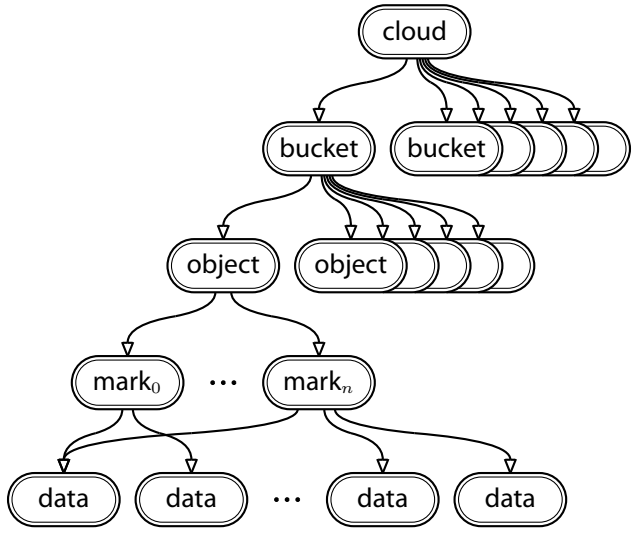

Figure 6.5: A example `cloud` DAO mimicking Amazon S3 / Google Data Cloud logic layout. In this instance one of the objects has $n + 1$ versions, mark$_0$ through mark$_n$. Note that `data` DAO can be shared amongst versions.

As stated above, Amzon S3 and Google Data Cloud can keep object buckets close to the applications running on *EC2* (Amazon) or *App Engine* (Google). To enable similar processing by the bucket load, proper replication strategies have to be applied to `bucket` DAOs. Data proximity allows data-handling systems like Google's (nonrelational) BigTable and Amazon's RDS (Relational Database Service), to provide the application with high-performing data selection. The augmented micro-object system would allow high performing data selection as well. For example, adding a `select` DAO to a `queue` DAO would trigger applications close to the home servers to start a selection, as described in the `select` DAO, on the locally hosted objects and cluster the matching ones to a `result` DAO. Especially given the encryption overhead, such an implementation would probably be slower than the Amazon and Google methods. However, the micro-object version would scale to many nodes and could even run in parallel to data entry.

Similarly, a lot of typical data cloud functionality could be offered by the augmented micro-object system, though, usually a special DAO has to be composed.

Using similar augmentation and composition a comparison to the commu-

nication part of the Hadoop MapReduce paradigm can also be made. Let us colloquially define Hadoop MapReduce as a framework to supply applications with a fault-tolerant way to process in parallel, massively distributed data. Note again how much overlap there is with the micro-object framework. Basic micro-objects are more primitive than the Hadoop MapReduce object, the file. As with the cloud-store comparison, the Hadoop MapReduce framework is usually run in conjunction with HDFS (Hadoop Distributed File System). In contrast to the cloud-store comparison, there is no need to mimic a distributed file system for standard DAOs can take the role of data distributor. Figure 6.6 shows an `eq-join` DAO modeled after the Hadoop MapReduce paradigm. The

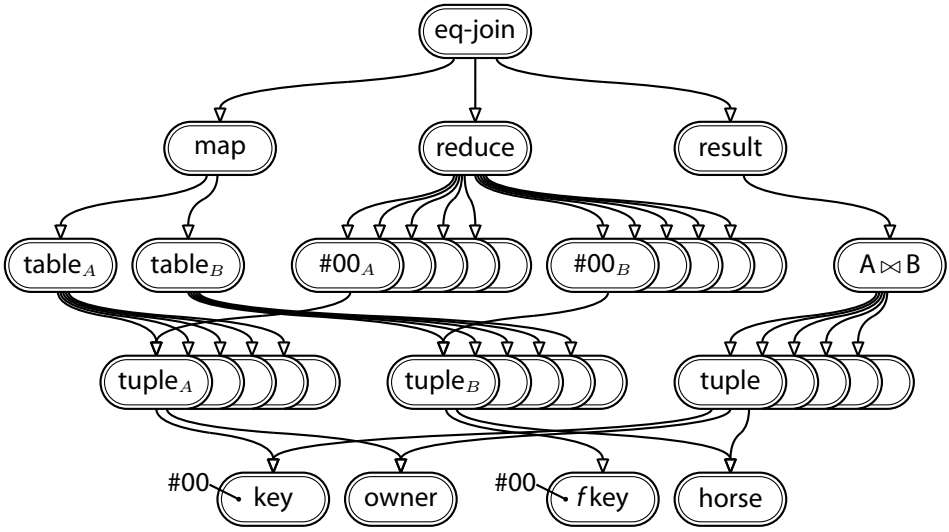

Figure 6.6: An `eq-join` DAO modeled after the Hadoop MapReduce paradigm. It holds a map, reduce and result DAO, each holding tables of tuples. Map applications have to cluster each `tuple` DAO to its designated hash table. Reduce applications have to join the corresponding hash tables. In this figure two tuples are shown to have matching keys. Therefore, they are mapped to matching hash tables and reduced to a <key, owner, horse> `tuple` DAO. Note the reuse of data nodes.

`eq-join` DAO holds a map, reduce and result DAO, each holding tables of tuples. Tuples hold data objects. In the beginning, the `map` DAO is initialized by adding the tables to be joined, table$_A$ and table$_B$. The `reduce` DAO is initialized by adding an empty hash table object per hash value per table. The `result` DAO

is initialized by adding a single empty table. Assuming several map and reduce applications are waiting for additions to some queue DAO, the eq-join process starts when the newly created `eq-join` DAO is added to this queue. The map applications find the tables in the `map` DAO and each start processing a subset of tuples, preferably "nearby" tuples. For each tuple a hash is calculated and based on the hash, the tuple is clustered to its corresponding hash table hanging off the `reduce` DAO. For every matching pair of hash tables, for example $\#00_A$ and $\#00_B$, a reduce application joins every new tuple arriving with all existing tuples of the opposing table. The results, if any, are clustered to the table of the `result` DAO. As with the previous comparison, performance is dependent on the encryption chosen as well as on the network connecting the applications as well as on the replication policies used. Unlike similar Hadoop MapReduce implementations [13], there is no need for separate map and reduce phases. Like the previous comparison, the whole process could be run in parallel with the input process. Note that this comparison is just a conceptual one. Hadoop MapReduce features important things like trackers to mitigate failure. Besides, any serious distributed database system based on micro-objects would probably feature back linked micro-objects to provide an easy way to find all tuples containing a given foreign key. This would make the above-mentioned mapping phase superfluous.

6.2.3 Future Work

As expected this adventure is far from over. Both unified messaging and micro-objects can be researched further. Though economic and political motivations will hinder adoption of a unified messaging system improving or even re-writing the existing implementation can give further insight on the emergent behaviour of super-sized messaging systems. An an industrial strength implementation should be developed that can be used to research replication policies, incidental-online devices, and usability. Such an implementation would probably necessitate an expansion of the current implementation of the micro-object system. But even in its own right, the micro-object system should be further developed. Most notably the available replication and security policies should be extended. It would be interesting to find a distribution vehicle, for example the Apache server, that would make it very easy to activate a multitude of cloud servers. This could then be used to study replication policies. It might also be interesting to research other uses of the micro-object system like the micro-object based sockets as discussed in Chapter 4 or a MapReduce-like application as mentioned above.

6.2.4 Summary

Unified messaging as defined in this thesis is based on the taxonomy of messaging systems also defined in the thesis. It aims to unify (most) all services offered by legacy messaging systems by replacing their implementations with one new (unified) implementation. This is a novel approach to messaging. To arrive at a manageable, secure and scalable system, the unified messaging system is split in two layers. The top layer is designed to deliver user-to-user messages in a variety of ways. The bottom layer delivers machine-to-machine messages in a novel way featuring a system based on the notion of the micro-object and direct data addressing. The micro-object system as defined in this thesis takes the visibility of partial failures to its logical conclusion and tries to make partial failures apparent not transparent. It features a unique way of constructing, protecting and diffusing distributed application objects.

Appendix A

Source Code

In Section 3.3 some code excerpts were given in figure 3.9, 3.10, 3.11, and 3.12. Below is the complete code. The lines that were included in the excerpts are in boldface. Note that this example code is optimized to produce nice excerpts, not efficiency.

```
/*
 * (c) Copyright 2005 Jan-Mark S. Wams
 *
 * This file is part of the micro-object middleware.
 *
 * The micro-object middleware is free software; you can redistribute
 * it and/or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * The micro-object middleware is distributed in the hope that it will
 * be useful, but WITHOUT ANY WARRANTY; without even the implied
 * warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
 * See the GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with the micro-object middleware; if not, write to the Free
 * Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
 * 02110-1301 USA
 */

/* Thise file contains some simple examples for using the micro-object
 * layer. Sniplest of these examples can be found in my dissertaion.
 */

#include <stdio.h>
#include <stdlib.h>
```

```
#include <inttypes.h>
#include <strings.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

#include "mo.h"

#define check(code)   if (!(code)) quit(#code, __FILE__, __LINE__);

/* Profiles. */
static void set_msg(bfer_t *);
static void get_msg(bfer_t *, bfer_t);
static void do_chat(bfer_t);
static void quit(char *, char *, int);
static void wait_chat(bfer_t);
static void busy_chat(bfer_t);
static void fprintmo(mo_t, tken_t, void *);
static void pr_cluster(mo_t, cter_t);

/* Main. Call the example functions. Note this function does not
 * have any variable of the mo_t type.
 */
int main(int argc, char **argv)
{
  bfer_t  bfer_radix_token, bfer_message;
  char   *r_store;

  /* Alloc ADT's. */
  check( bfer_alloc(&bfer_radix_token) );
  check( bfer_alloc(&bfer_message) );

   /*****************************/
  /* Demonstrate first example. */
 /*****************************/

  /* Create a new micro-object with a "Hello World!" payload.
   * Return its token in radix64 buffer.
   * Get and print the radix token as a string.
   */
  set_msg(&bfer_radix_token);
  check( bfer_getref_store(&r_store, bfer_radix_token) );
  printf("Radix token is '%s'\n", r_store);

  /* Get a copy from this micro-object,
   * get the payload and print it.
   */
  get_msg(&bfer_message, bfer_radix_token);
  check( bfer_getref_store(&r_store, bfer_message) );
```

```
  printf("The message is '%s'\n", r_store);

   /**********************************/
  /* Demonstrate the second example. */
 /**********************************/

  /* Chat using callback function. */
  printf("Do your thing, type ^D to quit.\n\n");
  do_chat(bfer_radix_token);


   /*********************************************************/
  /* Demonstrate simple example, not in my dissertation. */
 /*********************************************************/

  /* Chat using busy wait. */
  printf("Do your thing, type <ENTER> to refresh, ^D to quit.\n\n");
  wait_chat(bfer_radix_token);

   /**********************************/
  /* Demonstrate the third example. */
 /**********************************/

  /* Chat using busy wait. */
  printf("Do your thing, type ^D to quit.\n\n");
  busy_chat(bfer_radix_token);

  /* Dealloc ADT's. */
  check(bfer_free(&bfer_radix_token) );
  check(bfer_free(&bfer_message) );

  return 0;
}


/* Create a new micro-object, put Hello World! in the payload, and
 * return a buffer to its radix64-ified token.
 */
static void set_msg(bfer_t *p_bfer)
{
  mo_t   mo;
  plod_t plod;
  xpir_t xpir;
  psec_t psec;
  csec_t csec;
  tken_t tken;
  /* Allocate ADTs. */
  check( mo_alloc(&mo) );
  check( plod_alloc(&plod) );
```

```
  check( xpir_alloc(&xpir) );
  check( psec_alloc(&psec) );
  check( csec_alloc(&csec) );
  check( tken_alloc(&tken) );

  /* Fill the components for the micro-object. */
  check( plod_put_string(&plod, "Hello world!"));
  check( xpir_put_days_to_live(&xpir, 1));

#if 0
  bfer_key_pair_from_phrase(&key_pair, "Some Phrase.");
  psec_put(&psec, PSEC_ASYM_KEY, key_pair);
  csec_put(&csec, CSEC_NONE, NULL);
#endif

  /* Create the micro-object. */
  check( mo_create_new(&mo, xpir, plod, psec, csec) );

  /* Fill the output buffer with the radix64 of the token. */
  check( mo_get_tken(&tken, mo) );
  check( tken_to_radix64_bfer(p_bfer, tken) );

  /* Deallocate ADTs (in random order). */
  check( xpir_free(&xpir) );
  check( csec_free(&csec) );
  check( plod_free(&plod) );
  check( mo_free(&mo)     );
  check( tken_free(&tken) );
  check( psec_free(&psec) );
}


/* Get a (radix64 encoded) token, get a local copy of the micro-object,
 * extract and return a copy of its payload.
 */
static void get_msg(bfer_t *p_bfer_message, bfer_t bfer_radix_token)
{
  mo_t   mo;
  plod_t *r_plod;
  psec_t psec;
  csec_t csec;
  tken_t tken;

  /* Allocate ADTs. */
  check( mo_alloc(&mo) );
  check( psec_alloc(&psec) );
  check( csec_alloc(&csec) );
  check( tken_alloc(&tken) );
```

```
  check( tken_from_radix64_bfer(&tken, bfer_radix_token) );
#if 0
  bfer_key_pair_from_phrase(&key_pair, "Some Phrase.");
  psec_put(&psec, PSEC_ASYM_KEY, key_pair);
  csec_put(&csec, CSEC_NONE, NULL);
#endif

  /* Get a copy, get the payload, copy its buffer.
   */
  check( mo_create_copy(&mo, tken, psec, csec) );
  check( mo_getref_plod(&r_plod, mo) );
  check( plod_get_bfer(p_bfer_message, *r_plod) );

  /* Deallocate ADTs (in random order). */
  check( csec_free(&csec) );
  check( mo_free(&mo) );
  check( tken_free(&tken) );
  check( psec_free(&psec) );
}


void do_chat(bfer_t bfer_radix_mo_channel)
{
  mo_t   mo_line, mo_channel;
  plod_t plod;
  psec_t psec;
  csec_t csec;
  tken_t tken;
  cter_t cter_tracker;
  char   line[1024];
  xpir_t xpir;

  /* Allocate ADTs. */
  check( mo_alloc(&mo_line) );
  check( mo_alloc(&mo_channel) );
  check( cter_alloc(&cter_tracker) );
  check( plod_alloc(&plod) );
  check( psec_alloc(&psec) );
  check( csec_alloc(&csec) );
  check( tken_alloc(&tken) );
  check( xpir_alloc(&xpir) );

  check( tken_from_radix64_bfer(&tken, bfer_radix_mo_channel) );
#if 0
  bfer_key_pair_from_phrase(&key_pair, "Some Phrase.");
  psec_put(&psec, PSEC_ASYM_KEY, key_pair);
  csec_put(&csec, CSEC_NONE, NULL);
#endif
```

```
  /* Get a copy of the channel mo, and reuse the expir date.
   */
  check( mo_create_copy(&mo_channel, tken, psec, csec) );
  check( mo_get_xpir(&xpir, mo_channel) );
#if 0
  mo_put_replication(&mo_channel, FLOODING);
#endif

  /* Start forwarding the incomming tokens. */
  check( mo_cter_clbk(&mo_channel, &cter_tracker, fprintmo, stdout) );

  /* Create a prompt. */
  strcpy(line, "1>> ");

  /* Direct all input lines to channel. */
  while(fgets(line + 4, sizeof(line) - 4, stdin)) {
    plod_put_string(&plod, line);
    xpir_uinc(&xpir);
    mo_create_new(&mo_line, xpir, plod, psec, csec);
    mo_cter_add_mo(&mo_channel, mo_line);
  }

  /* Stop forwarding the incomming tokens.  */
  check( mo_cter_clbk(&mo_channel, NULL, NULL, NULL) );

  /* Deallocate ADTs (in random order). */
  check( cter_free(&cter_tracker) );
  check( psec_free(&psec) );
  check( xpir_free(&xpir) );
  check( tken_free(&tken) );
  check( mo_free(&mo_line) );
  check( mo_free(&mo_channel) );
  check( csec_free(&csec) );
  check( plod_free(&plod) );
}


#include <signal.h>

/* Don't do anything. */
static void handler(int sig) { /* nop */}

/* Return 0 if EOF. */
static int read_noblock(char *buf, int size)
{
  void *res;

  signal(SIGALRM, handler);
  errno = 0;
```

```
  alarm(1);
  res = fgets(buf, size, stdin);
  alarm(0);

  if (NULL != res)
    return 1;

  if (EINTR == errno) {
    buf[0] = '\0';
    return 1;
  }

  /* EOF seen, or error other than EALARM. */
  return 0;
}


/* Simple nontrheaded, noncallback, blocking version.
*/
static void wait_chat(bfer_t bfer_radix_mo_channel)
{
  mo_t   mo_line, mo_channel;
  plod_t plod;
  psec_t psec;
  csec_t csec;
  tken_t tken, *r_tken;
  cter_t *r_cter;
  char   line[1024];
  xpir_t xpir;
  enum_t e;

  /* Allocate ADTs. */
  check( mo_alloc(&mo_line) );
  check( mo_alloc(&mo_channel) );
  check( plod_alloc(&plod) );
  check( psec_alloc(&psec) );
  check( csec_alloc(&csec) );
  check( tken_alloc(&tken) );
  check( xpir_alloc(&xpir) );

  check( tken_from_radix64_bfer(&tken, bfer_radix_mo_channel) );
#if 0
  bfer_key_pair_from_phrase(&key_pair, "Some Phrase.");
  psec_put(&psec, PSEC_ASYM_KEY, key_pair);
  csec_put(&csec, CSEC_NONE, NULL);
#endif

  /* Get a copy of the channel mo, and reuse the expir date.
   */
```

```
  check( mo_create_copy(&mo_channel, tken, psec, csec) );
  check( mo_get_xpir(&xpir, mo_channel) );
  check( xpir_usec_add(&xpir, 2000L) );    /* Scip experiment 1 messages. */
#if 0
  mo_put_replication(&mo_im, FLOODING);
#endif

  /* Create a prompt. */
  strcpy(line, "2>> \n");

  do {
    /* If there is a true message, put it on the channel. */
    if ('\n' != line[4]) {
      check( plod_put_string(&plod, line) );
      check( xpir_uinc(&xpir) );
      check( mo_create_new(&mo_line, xpir, plod, psec, csec) );
      check( mo_cter_add_mo(&mo_channel, mo_line) );
    }

    /* Print entire cluster. */
    mo_getref_cter(&r_cter, mo_channel);
    cter_enum_create(&e, *r_cter);
    while (cter_enum_getref_next_tken(&e, &r_tken, NULL))
      fprintmo(mo_channel, *r_tken, stdout);

  }  while (fgets(line + 4, sizeof(line) - 4, stdin));


  /* Deallocate ADTs (in random order). */
  check( psec_free(&psec) );
  check( xpir_free(&xpir) );
  check( tken_free(&tken) );
  check( mo_free(&mo_line) );
  check( mo_free(&mo_channel) );
  check( csec_free(&csec) );
  check( plod_free(&plod) );
}


/* Added a bit interactivety by using a nonblocking way to read the keyboard.
 * Note that it outputs ANSI terminal (VT100 compatible) codes.
 */
static void busy_chat(bfer_t bfer_radix_mo_channel)
{
  mo_t   mo_line, mo_channel;
  plod_t plod;
  psec_t psec;
  csec_t csec;
  tken_t tken;
```

```
  cter_t *r_cter;
  char   line[1024];
  xpir_t xpir;

  /* Allocate ADTs. */
  check( mo_alloc(&mo_line) );
  check( mo_alloc(&mo_channel) );
  check( plod_alloc(&plod) );
  check( psec_alloc(&psec) );
  check( csec_alloc(&csec) );
  check( tken_alloc(&tken) );
  check( xpir_alloc(&xpir) );

  check( tken_from_radix64_bfer(&tken, bfer_radix_mo_channel) );
#if 0
  bfer_key_pair_from_phrase(&key_pair, "Some Phrase.");
  psec_put(&psec, PSEC_ASYM_KEY, key_pair);
  csec_put(&csec, CSEC_NONE, NULL);
#endif

  /* Get a copy of the channel mo, and reuse the expir date.
   */
  check( mo_create_copy(&mo_channel, tken, psec, csec) );
  check( mo_get_xpir(&xpir, mo_channel) );
  check( xpir_usec_add(&xpir, 1000L) );    /* Scip experiment 1 messages. */
#if 0
  mo_put_replication(&mo_channel, FLOODING);
#endif

  /* Create a prompt. */
  strcpy(line, "3>> ");

  printf("\033[2J"); /* Clear screen. */
  printf("\033[%d;%dH", 24, 1); /* Goto row, column (24,1). */

  for(;;) {
    /* Print cluster so far. */
    mo_getref_cter(&r_cter, mo_channel);
    pr_cluster(mo_channel, *r_cter);

    /* Get user input, but don't wait to long. */
    if (!read_noblock(line + 4, sizeof(line) - 4)) break;

    if ( '\0' != line[4]) {
      check( plod_put_string(&plod, line) );
      check( xpir_uinc(&xpir) );
      check( mo_create_new(&mo_line, xpir, plod, psec, csec) );
      check( mo_cter_add_mo(&mo_channel, mo_line) );
    }
```

```
  }

  printf("\033[2J"); /* Clear screen. */
  printf("\033[%d;%dH", 24, 1); /* Goto row, column (24,1). */

  /* Deallocate ADTs (in random order). */
  check( psec_free(&psec) );
  check( xpir_free(&xpir) );
  check( tken_free(&tken) );
  check( mo_free(&mo_line) );
  check( mo_free(&mo_channel) );
  check( csec_free(&csec) );
  check( plod_free(&plod) );
}


/* --- Misc functions. --- */

/* The function of no return.
*/
static void quit(char *code, char *fname, int lineno)
{
  fprintf(stderr, "%s:%d: \"%s\" failed (sorry).\n", fname, lineno, code);
  exit(1);
}

/* Print a whole cluster. (Use fprintmo).
 */
static void pr_cluster(mo_t mo_channel, cter_t cter)
{
  enum_t e;
  tken_t *r_tken;

  printf("\0337"); /* Save cursor. */

  printf("\033[H"); /* Goto top left. */

  /* Print entire cluster. */
  cter_enum_create(&e, cter);
  while (cter_enum_getref_next_tken(&e, &r_tken, NULL)) {
    printf("\033[2K"); /* Erase line. */
    fprintmo(mo_channel, *r_tken, stdout);
  }

  printf("\0338"); /* Unsave cursor. */

}

/* Print the payload of the micro-object of the new token.
```

```
*/
static void fprintmo(mo_t mo_channel, tken_t tken_message, void *user_data)
{
  mo_t    mo_message;
  FILE   *fp = (FILE *)user_data;
  plod_t *r_plod_message;
  bfer_t *r_bfer_message;
  char   *char_message;
  psec_t  psec;
  csec_t  csec;

  /* Allocate ADTs. */
  check( mo_alloc(&mo_message) );
  check( psec_alloc(&psec) );
  check( csec_alloc(&csec) );

#if 0
  bfer_key_pair_from_phrase(&key_pair, "Some Phrase.");
  psec_put(&psec, PSEC_ASYM_KEY, key_pair);
  csec_put(&csec, CSEC_NONE, NULL);
#endif

  /* Get a copy, get the payload's buffer's store.
   */
  check( mo_create_copy(&mo_message, tken_message, psec, csec) );
  check( mo_getref_plod(&r_plod_message, mo_message) );
  check( plod_getref_bfer(&r_bfer_message, *r_plod_message) );
  check( bfer_getref_store(&char_message, *r_bfer_message) );

  /* Print it out! */
  fprintf(fp, "%s", char_message);

  /* Deallocate ADTs (in random order). */
  check( csec_free(&csec) );
  check( mo_free(&mo_message) );
  check( psec_free(&psec) );
}
```

Index

ANSI-41 MAP, 24
Apparent failure, 61

Bean, 159
    Stateful, 159
    Stateless, 159
Bearer channels, 24
Blog, *see* Weblog
Blogging, *see* Weblog
Botnet, 31

Cloud server, 62
Content list, 97
    Fragmentation value, 103
    Replacement value, 100
Content-based networking, 160
Content-centric networking, 160
CORBA, 158
    IDL, 158
Could computing, 163

Data addressing
    Content-based, 161
    Pipe-based, 161
    Subject-based, 161
    Token-based, 161
Data cloud, 163
Ditto-list, 75
Domain Name System
    Query example, 13

E-mail, 9--14
    At symbol (@), 13
    End-point delivery, 26
    Mailbox, 10
    Naming, 13--14
    Point-to-point communication, 16
    Post office protocol, 12
    Remote access, 12
    Resolving a name, 13--14
    Webmail, 12
eBay, 36
EJB
    Containers, 159
    Entity bean, 159
    Message-driven bean, 159
    Stateful bean, 159
    Stateless bean, 159
Enterprise Java Beans, 159
Entity bean, 159

Facebook, 36
    Weblog, 36
Flat rate payment, 35

GSM MAP, 24

IDL, 158
Instant messaging, 17--21
    Channel, 18
    Chat room, 19
    Chat session, 19
    Naming, 21
    Presence, 19--21
    Term-talk, 17
IS-41 MAP, 24

Large scale messaging, 8
Linear rate payment, *see* Flat rate payment
LinkedIn, 36

Massive messaging, 8
Maximum adaptability, 41
Message-driven bean, 159
Messaging Service
    Deleting messages, 44
    Name space, 43
    Presence, 43
    Security, 43
Messaging system, 7

Address dimension, 33
Audience dimension, 33
Direction dimension, 33
Taxonomy, 32--35
Time dimension, 32
Micro-object, 59--108
Additional replication, 71
Basic replication, 71
Cluster, 63
Definition, 62--64
Eventual consistency, 72
Expiration date, 67
Hello World example, 90
Heterogeneous replication, 72
Home location, 62
Payload, 62
Token, 62
Mix-networks, 70
MO, 62
Mo_create_copy(), 80
Mo_create_new(), 78
Mo_cter_add_tken(), 88
Mo_get_cter(), 84
Mo_get_tken(), 79
Mo_put_cter_clbk(), 85
Mo_put_repl(), 82
Moping, 141--142
MySpace, 36

Network News Transfer Protocol
Extensions to, 15
Flooding, 16
News feed, 15
Newsgroup, 14

Object cloud, 62

Pidgin, 114
Ping, 141
Predicate addressing, 160
Preferred mail server, 13
Presence, *see* Instant messaging, Presence, 116

SMS, 24--28
Area code, 27
Base station, 27
Cellular structure, 27
Country code, 27
End-point delivery, 24
Global title, 28
Message length, 25
Mobile subscriber identity number, 27
Naming, 27--28
Network code, 27
Numbering plan indicator, 28
Roaming end point, 27
Roaming user, 24
Store-and-forward, 24
Subscriber number, 27
Type of service, 28
Social network messaging, 36
Spam, 30
Spim, 126
SQLite, 115
SS5, 24
In-band signaling, 24
SS7, 25
Out-of-band data, 25

Tomlinson, Ray, 9
Twitter, 36

UMS server, 115
Unified Messaging
immutability, 46
moderation, 51
output moderating, 52
Unified messaging, 28, 42--50
Adaptability, 41
Definition, 42
Goals of, 45--46
home location, 48
Input moderating, 51
Message-pull, 48
Model of, 45
Post-key, 46
Read-key, 46
Say-all, hear-all, 50
System, 113--132
Target, 46
Terminology, 46
TISM, 46
Total moderating, 52
UMS-tuple, 47
USENET, 14--17
Articles, 14
Body, 14
Cross posting, 14
Header, 14
Hierarchical naming, 16

Naming, 15--17
Newsgroup, 14

Weblog, 21--23
Blog service providers, 22
Blogger, 22
Blogosphere, 22
CMS backend, 22
CMS font-end, 22
Content management system, 22
RSS, 22
Webmail, 12

YouTube, 36

Zoosk, 36

List of Acronyms

ADT . . . . . Abstract Data Type An ADT is a specification of a set of related data items and the operations that can be performed on those items. It is Abstract in the sense that it separates the implementation from the specification. It is the cornerstone of programming. *Page 77*

AIM . . . . . AOL Instant Messenger AIM is an instant message system allowing users to communicate by exchanging text messages in realtime. *Page 49*

AOP . . . . . Aspect Oriented Programming AOP is utilizing a way of modularization of a program to separate concerns, specifically cross-cutting concerns. *Page 60*

API . . . . . Application Programming Interface An API is a computer language definition, how one layer in a computer system can communicate with the next, usually consisting of a list of function calls with their parameters and return types. *Page 62*

ARPANET . . . . . Advanced Research Project Agency Network ARPANET is one of the precursors to the Internet, ARPANET was a large wide-area network created for testing purposes by the United States Defense ARPA in 1969. *Page 9*

ASCII . . . . . American Standard Code for Information Interchange ASCII is a character encoding. *Page 25*

BBM . . . . . BlackBerry Messenger BBM is a proprietary instant messaging application included on BlackBerry devices allowing user-to-user as well as chat-group messaging. It allows exchange of text, image, audio, and other type of messages. Users are addressed using a PIN code, a QR code, or an E-mail address. *Page 49*

BBS . . . . . Bulletin Board System A BBS is a computer with one or more modems, that allows users to exchange messages. It also allowed users to upload and download data. There were many BBSes, run by amateurs during the 1980's and the first half of the 1990's. Two early BBS are Charles Oropallo's Access-80 (1977) and Ward Christensen's CBBS (1979). *Page 52*

BITNET . . . . . Because It's Time NETwork (earlier: Because It's There NETwork) BITNET started out as the CUNY (City University of New York) and Yale University messaging network in 1981. BITNET users migrated to Internet based messaging.

BOTNET . . . . . roBOT NETwork A botnet is a collection of hacked computers that are connected to the Internet. A botnet usually has one or more herders that are able to control the actions of the so called bots. Typical instructions would include attacking a website and sending SPAM messages. *Page 31*

CAPTCHA . . . . . Completely Automated Public Turingtest to tell Computers and Humans Apart A captcha is a response test to force human interaction in an automated process. For example an word is displayed and the user is expected to enter that word. The word usually is displayed in a manner that is hard to read for computer programs, but still legible enough for humans. *Page 128*

CDMA . . . . . Code Division Multiple Access CDMA is a technology to allow multiple access to a (cellular) network. CDMA does not assign a specific frequency band to each user (as does FDMA) instead, every channel uses full spread of the frequency domain and a channel identifier. CDMA is also referred to as spread-spectrum. See also FDMA, and RF. Note, CDMA is not to be confused with WCDMA, cdmaOne, and CDMA2000, because those are CDMA based mobile phone standards. *Page 27*

cHTML . . . . . Compact HTML cHTML is a, HTML like, content description language for i-mode devices describing one screen per file. cHTML is becoming less relevant as devices capable of handling (full-blown) HTML and WML become more widespread. See also HTML, WML and i-mode.

CIMD . . . . . Computer Interface to Message Distribution CIMD is a data exchange protocol, developed by Nokia, to communicate with SMSCs. See also SMSC. *Page 8*

CMS . . . . . Content Management System A CMS helps separate content form format. A CMS typically has a back-end that is used to edit the content and formatting, and a front-end that allows users to access the (formatted) content. *Page 22*

CORBA . . . . . Common Object Request Broker Architecture CORBA is a middleware layer to develop distributed application, featuring an interface definition language that is independent of development platform. The basis of CORBA is remote procedure call. *Page 60*

CSD . . . . . Circuit Switched Data CSD is a network technology that offers communicating partners a temporary direct connection with each other over limited shared resources. A CSD network has connection setup and release (or teardown) procedures. POTS is a CSD network. Packet Switching (e.g., the Internet and GPRS) is the opposite of CSD. See also POTS and GPRS. *Page 25*

CTI . . . . . Computer-Telephony Integration CTI is a system that allows interactions between communication via the telephone system and via the Internet. Contact channels like E-mail, SMS, fax, voicemail, can be used and managed through both the Internet and the telephone system. *Page 156*

DAO . . . . . Distributed Application Object A DAO is an collection of related data and functions in an application program. It need not be an object in the stricter sense of object oriënted programming (that is, an instance of a class), but it can be. *Page 94*

DIG . . . . . Domain Information Groper Dig is a program that finds the DNS IP address for a given hostname or (with the -x flag) vice versa. *Page 13*

DNS . . . . . Domain Name System The DNS is an overlay network of servers that translate names, like `www.cs.vu.nl`, into IP addresses like `130.37.20.20`. The DNS is like a giant telephone book. However, the DNS is a distributed system, allowing for constant updates. *Page 13*

DoS . . . . . Denial of Service A DoS attack is an attempt to make a computer resource unavailable to its rightful users. It is like a thousand grannies jumping the queue as you await your turn to use the bathroom. *Page 105*

DSM . . . . . Distributed Shared Memory DSM is virtual memory that is shared over a network by several hosts. To each host, reading and writing to this virtual memory appears to be just like reading and writing local shared memory as if the sharing took place at one location. *Page 107*

EJB . . . . . Enterprise Java Beans EJB is a collection of server-side services that are provided by the EJB runtime environment. Typically, concurrent data access and security is provided by the EJB container with some degree of networking transparency. EJB in it's core facilitates client/server communication. *Page 60*

E-MAIL . . . . . Electronic Mail E-mail is the largest homogeneous messaging system in the world. *Page 8*

EMS . . . . . Enhanced Messaging Service (also: Enhanced Messaging Service) EMS is extension to SMS for sending and receiving formatted text, images, sound clips, and ring tones. See also SMS. *Page 26*

FDMA . . . . . Frequency Division Multiple Access FDMA is a technology to allow multiple access to a (cellular) network. FDMA will cut a frequency domain into several frequency bands or channels. It is particularly commonplace in satellite communication and the GSM network. See also GSM, and RF. *Page 27*

FTP . . . . . File Transfer Protocol FTP is a client/server protocol for transferring files over TCP networks. It is mostly used to up- or download files from a local client to a remote server. More specialized usage includes transferring files directly between remote servers. *Page 61*

GNUPG . . . . . GNU Privacy Guard GnuPG can encrypt E-mail messages using asymmetric encryption. Each individual user generates a private, public key pair. The resulting public keys have to be exchanged with other users. GnuPG provides a way to secure content as well as a way to sign content. *Page 122*

GPRS . . . . . General Packet Radio Service GPRS is a packet switched GSM overlay network. Because it is packet switched it allows more users per network and it is better suited for data applications such messaging. Practically GPRS offers full Internet access to roaming cellular devices. See also GSM.

GSM . . . . . Global System for Mobile Communications GSM is the leading cellular systems in Europe and Asia. See also TDMA. *Page 27*

GUI . . . . . Graphical User Interface A graphical user interface (pronounced "gooey") is a method of interacting with a computer through a metaphor of graphical images in addition to text, usually involving a mouse, keyboard, and screen. *Page 115*

HLR . . . . . Home Location Registers A HLR is a SS7 database containing up-to-date information on cell phones. It is much like a POTS database, but for the SS7 to function, the HLR contains additional information to accommodate roaming. See also POTS, glxrefNSS, IMSI, and SS7. *Page 27*

HTML . . . . . HyperText Markup Language HTML is a language used to create multimedia documents. A typical HTML page contains text, images and links to other HTML documents. *Page 12*

HTTP . . . . . HyperText Transfer Protocol HTTP is the predominant protocol used for exchange of WWW pages. See also WWW.

ICQ . . . . . I seek you ICQ is a program, created by the Israeli company Mirabilis int 1996. In 1998 ICQ was bought by AOL, and it is slowly merged with AOL's other IM system, AIM. See also AIM and IM. *Page 17*

IDL . . . . . Interface Definition Language An IDL is special language to describe how two layers in a computer system communicate. Usually IDL specifications are compiled into an API and part of the implementation of the API. See also API. *Page 60*

IM . . . . . Instant Messaging IM allows individual users to exchange short text statements in real time. It could be compared to having a telephone conversation where talking has been replaced by typing on a keyboard. *Page 9*

I-MAIL . . . . . Information mail i-mail is a messaging system for i-mode devices. See also i-mode. *Page 8*

IMAP . . . . . Internet Message Access Protocol IMAP is a text based protocol for retrieving E-mail from a server machine to a client machine. IMAP allows a user specify a subset of messages to download. Typically, messages are not deleted from a server after they have been copied to a client machine. See also POP. *Page 12*

I-MODE . . . . . Information mode (also: Internet mode) i-mode is NTT DoCoMo's mobile Internet access system.

IMSI . . . . . International Mobile Subscriber Identity A IMSI is a unique (fifteen or less digits) number used to identify a GSM (and UMTS) cellular device. A IMSI has three parts: the country code (MCC), the network code (MNC), and a unique subscriber number (MSIN) and is used to lookup information in the HLR. See also HLR. The IMSI conforms to the ITU E.212 numbering standard. *Page 27*

IP . . . . . Internet Protocol IP is a datagram delivery and addressing protocol, with store and forward delivery. The routing of an IP packet, as IP datagrams are sometimes called, follows one simple rule; If the IP address is on the local network, send it directly to the destination, otherwise, send it to the router. It is up to the router to send IP packets to (a router closer to) their destination. The IP protocol is used for computer to computer (or host to host, in Internet speak) communication. Usually for process-to-process communication a higher protocol with sub addressing is used. See also UDP and TCP. *Page 14*

IRC . . . . . Internet Relay Chat IRC is many-to-many (including one-on-one) IM program. IRC was created by the Fin Jarkko Oikarinen in 1988 and it was loosely based on BRC (BITNET Relay Chat). See also BITNET, IM, and ICQ. *Page 17*

ISDN . . . . . Integrated Services Digital Network ISDN is an international communications protocol for sending voice and data over digital or analog telephone lines with a bit rate of 64 Kbps. *Page 28*

ISP . . . . . Internet Service Provider An ISP is an organisation that provides Internet connectivity and services to individual users. Usually the users have to connect to an ISP via a modem over a phone or DSL line. Typically an ISP would run mail and web processes on the users behalf, allowing the user to be offline most of the time. *Page 12*

ITU . . . . . International Telecommunication Union The ITU is on of the oldest (founded in 1865) international organizations and its goal is to standardize and regulate international telecommunications and radio. Since 1947 the ITU has been an agency of the United Nations (UN). The ITU has three main suborganizations:

ITU-T (before 1992 called the CCITT) for telecommunications, ITU-R for radio communications, and ITU-D for development. *Page 25*

LAN . . . . . Local Area Network A LAN is a communication network that allows direct communication between parties over relative short distances. Compared to a WAN connection speeds are usually high and communication costs are usually low. See also WAN. *Page 140*

LRU . . . . . Least Recently Used A LRU caching algorithm, selects cached items to be replaced based on the time they have last been used. The general idea behind that, is that items that have a high probability of being used in the future, will have been recently used. Generally this LRU selection will outperform random selection. *Page 77*

MAP . . . . . Mobile Application Part MAP is a protocol for near real time communication between nodes in a cellular network. For example the MAP protocol is used to transfer information to and from the VLR and the HLR. See also VLR and HLR. *Page 24*

MD5 . . . . . Message Digest algorithm 5 MD5 is a deterministic algorithm that converts an arbitrary sized array of bytes into a fixed number of bits. In theory any change to the input, even a single bit, will result flip the output bits with an even chance. *Page 140*

MDA . . . . . Message Delivery Agent The MDA is responsible for moving messages from the MTA into the proper mailboxes. *Page 11*

MMS . . . . . Multimedia Message Service MMS is a WAP based cellular protocol for transmitting images, video clips, sound and text messages. See also WAP. *Page 8*

MO . . . . . micro-object A MO is a small simple distributed object that supported by the MO system. They are the building blocks of more complex distributed objects, most notably DAOs. See also DAO. *Page 62*

MSA . . . . . Message Submit Agent The MSA is the server that queues outgoing messages from MUAs, until they are ready to be sent out by the MTA. See also MTA and MUA. *Page 10*

MTA . . . . . Message Transfer Agent Server process to exchange incoming and outgoing E-mail. The MTA at a sender's site is responsible for removing messages that have been queued by the MSA, and transferring them to their destinations, possibly routing them across several other MTAs. At the receiving side, the MTA spools incoming messages, making them available for the See also MSA and MDA. *Page 10*

MUA . . . . . Mail User Agent A MUA is a user application for accessing and managing E-mail. Users often refer to a MUA as their E-mail client. The MUA provides the interface between the user and their MSAs, MDAs, and POP servers. See also MSA, MDA, and POP. *Page 10*

MX . . . . . Mail Exchange A MX record is a set of prioritized entries in the DNS database identifying the mail servers that are responsible for handling E-mail for that domain. See also DNS. *Page 13*

NNTP . . . . . Network News Transfer Protocol NNTP is used for exchanging USENET text messages. The official specification is RFC 977. See also USENET. *Page 15*

NSS . . . . . Network and Switching Subsystem (also known as: the GSM core network) The NSS is the circuit switched GSM core network, for traditional services like voice calls and SMS messaging.

NTA . . . . . News Transfer Agent A NTA is a server process that receives, stores, and forwards USENET news articles. See also USENET. *Page 15*

NUA . . . . . News User Agent A NUA is a client program that enables the user to create, send, receive, and read USENET news articles. See also USENET. *Page 15*

OOP . . . . . Object Oriented Programming OOP is a programming paradigm that uses objects to build applications. It utilizes several paradigms like: inheritance, modularity, polymorphism, and encapsulation. Building large application, often suffers from problems like namespace pollution, using OOP will solve most of these kind of problems. Many current programming languages have support for OOP. Note, OOP does not happen automatically simply because a language supports OOP, it is a way of designing and implementing. *Page 130*

PGP . . . . . Pretty Good Privacy PGP is a method to provide end-to-end (asymmetric) encryption and signing for messaging. Most commenly used for e-mail. It was created by Phil Zimmermann in 1991 but now has been formalized into the OpenPGP standard (RFC 4880). *Page 43*

POP . . . . . Post Office Protocol POP is a text based protocol for retrieving E-mail from a server machine to a client machine. Usually, once the messages have been copied to the client machine, they are deleted on the server machine. See also IMAP. *Page 12*

POTS . . . . . Plain Old Telephone Service POTS stands for the normal wired analogue telephone system as developed in 1876 by Alexander Graham Bell. Also referred to as PSTN. *Page 7*

PSTN . . . . . Public Switched Telephone Network PSTN is the international telephone system based on copper wires carrying analog voice data. Also referred to as POTS.

QoS . . . . . Quality of Service QoS originally stood for the probability that a packed switched network would a predetermined throughput level. The definition of QoS has been expanded to stand for the probability that any communication service delivers a given throughput, failure, or prefetch rate. *Page 68*

RDF . . . . . Resource Description Framework RDF allows the exchange of information on collections of music, books, news articles, or photos. RDF uses XML as an interchange syntax. Some consider RDF to be the precursor of RSS. See also XML and RSS.

RF . . . . . Radio Frequency RF is the frequency of an electromagnetic spectrum that propagates through space and has wave like properties. Scales range from Ultra Low Frequency (less than 3 Hz) up till Extremely High Frequency (300 GHz). *Page 27*

RMI . . . . . Remote Method Invocation Remote Method Invocation is the object-oriented equivalent of RPC. An application object is represented on the client side by a standin or stub object. Methods invoked on the stub are forwarded to a server for processing. Resulting values follow the reverse route. *Page 159*

RPC . . . . . Remote Procedure Call A RPC is a method for two computers to act as one. The caller on one computer sends arguments to the callee on another computer for processing and waits for the callee to send back the results. *Page 60*

RSS . . . . . Really Simple Syndication (also: Rich Site Summary and RDF Site Summary) RSS is an XML format for syndicating web content. See also XML and RDF. *Page 22*

SCCP . . . . . Signalling Connection Control Part SCCP is an ITU routing protocol (Q.713) for SS7 networks. See also SS7.

SCTP . . . . . Stream Control Transmission Protocol SCTP is a transport layer protocol on top of the IP layer. It is message-oriented (like UDP) featuring in-sequence delivery and congestion control (like TCP). SCTP is mainly used to carry PSTN signals over IP networks. See TCP, UDP, PSTN, SS7, and IP. *Page 26*

SDMA . . . . . Space Division Multiple Access SDMA is a technology to allow multiple access to a (cellular) network. SDMA will adapt the direction and the power of an RF signal based on the relative location of the receiving end. This minimizes RF interference and maximizes frequency band reuse. Most commonly used by GSM networks in combination with FDMA. See also RF, GSM, and FDMA. *Page 27*

SHF . . . . . Super High Frequency A RF band with a wavelength from 100mm till 10mm (3-30GHz). Used for cellular networks, wireless LAN, etc. *Page 27*

SIGTRAN . . . . . SIGnaling TRANsport SIGTRAN is a set of protocol extension to SS7. The most significant of these is SCTP for carrying PSTN signals over IP networks. See SCTP, PSTN, SS7, and IP. *Page 26*

SIM . . . . . Subscriber Identity Module A SIM (card) is a smart card for a cellular device used to identify and authenticate a user to the cellular network. The SIM can also contain other data, for example: device settings and user data (for example, phone numbers). *Page 27*

SMPP . . . . . Short Message Peer-to-Peer Protocol SMPP is a protocol for application to send short messages to a SMSC. See also SMSC and SMS. *Page 8*

SMS . . . . . Short Message Service SMS is a service for sending short text messages to cellular phones. SMS messages are typically less than 160 characters long. *Page 7*

SMSC . . . . . Short Message Service Center SMSC is a server that gets SMS text messages to the appropriate mobile device. See also SMS. *Page 25*

SMTP . . . . . Simple Mail Transfer Protocol SMTP is the predominant protocol for MTA to communicate. See also MTA. *Page 11*

SPAM . . . . . not the luncheon meat canned by Hormel Foods Spam was first identified on the Usenet, and got its name from a Monty Python sketch. The first infamous professional spammers was Laurence Canter, co-author of the book *How To Make A Fortune On The Information Superhighway*. *Page 10*

SPIM . . . . . spam over IM (also: instant spam) Spim is spam for IM systems. Spim messages are sent out by a human IM user simulator, called a spim-bot, to IM screen names that have been harvest off the Internet. A spimmer is the individual or organization responsible for sending the spim. *Page 31*

SQL . . . . . Structured Query Language SQL is a database query and management language developed in the 1970's, initially by IBM and Relational Software (the current Oracle Corporation). *Page 115*

SRV . . . . . SeRvice The DNS SRV record contains location data, for example hostname and port number, of servers for specified services. It is used, for example, by the Extensible Messaging and Presence Protocol. See also XMPP. *Page 19*

SS5 . . . . . Signaling System 5 SS5 was one of the precursors to SS7. As an ITU standard, it was once in widespread use, but has long since been replaced by SS7. See also ITU and SS7. *Page 24*

SS7 . . . . . Signaling System 7 (also known as: C7, number 7, and CCIS7) SS7 is an ITU circuit switching out-of-band data exchange (three layer) protocol stack for telephone networks at up to 64kbps. It was introduced in 1981 for use on the NSS. See also ITU, SCCP, and NSS. *Page 8*

SSL . . . . . Secure Sockets Layer is an encryption protocol to secure Internet communication. It features: peer negotiation to select one of the supported encryption algorithms, public-key encryption-based key exchange, certificate-based authentication, symmetric cipher-based session encryption. Note: SSL transformed into TLS (Transport Layer Security). *Page 121*

TCAP . . . . . Transaction Capabilities Application Part TCAP is a protocol for SS7 networks. It multiplexes connections by adding transaction IDs to messages. These transaction IDs are similar to TCP ports. See also TCP. TCAP is used by the MAP and other layers. See also MAP. According to the SS7 standard TCAP is part of the application layer. *Page 24*

TCP . . . . . Transmission Control Protocol TCP is a protocol designed for the Internet. In contrast to the IP protocol, that deals only with packets, TCP enables data streams. TCP adds several sorts of error correction and optimization to the service of IP. Most notably TCP can handle limited IP packet loss and out of order delivery of IP packets. Also, like UDP, TCP introduces the notion of a "port," allowing up to sixty five thousand distinct connections per IP address. See also UDP and IP. *Page 13*

TDMA . . . . . Time Division Multiple Access TDMA is a technology to allow multiple access to a (cellular) network. TDMA assigns each user a single frequency band and fixed interval time slots to use it. TDMA is used in combination with FDMA (and sometimes SDMA) by GSM networks to support eight data channels per frequency band. See also GSM, FDMA, SDMA, and CDMA. *Page 27*

TISM . . . . . Targeted Immutable Short Message a TISM is the generic name for a unit of exchanged data, also referred to by its generic name as message. *Page 46*

UA . . . . . User Agent A UA is an application that allows users to communicate with the underlying IM system. See also IM. *Page 18*

UBE . . . . . Unsolicited Bulk Email UBE is a synonym for spam: unsolicited messages. *Page 30*

UCE . . . . . Unsolicited Commercial Email UCE is a synonym for spam: unsolicited messages. *Page 30*

UDP . . . . . User Datagram Protocol UDP is a connectionless protocol that, like TCP, runs on top of IP networks. Unlike TCP, UDP provides no error correction to speak of. The main reason for using it rather than using IP directly, is its introduction of the notion of a "port," allowing up to sixty five thousand distinct sub addresses per IP address. See also TCP and IP. *Page 99*

UMS . . . . . Unified Messaging System A UMS is a system that mimics different messaging services and can be used to replace the services of all other messaging systems. *Page 42*

UNIX . . . . . Uniplexed Information and Computing Service UNIX is a computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Brian Kernighan. The original UNIX has split into various commercial and nonprofit branches. *Page 137*

URI . . . . . Uniform Resource Identifier A URI is an object (or resource) name (or identifier) possibly including location information, within a given namespace. A URI that does not include location information is referred to a URN, otherwise an URI is referred to as URL. See also URN and URL.

URL . . . . . Uniform Resource Locator A URL uniquely identifies an object on the WWW, it also identifies the object's location. A URL can contain a scheme (or domain), host, port, path, parameters, and values. However, in practice most URLs are of the scheme://host/ or scheme://host/path format. The set of URLs is a proper subset of the set of URIs. See also WWW and URI. *Page 68*

URN . . . . . Uniform Resource Name A URN is a URI that identifies a resource by name in a particular namespace, without including location information. The set of URNs is a proper subset of the set of URIs. See also URI.

USENET . . . . . UNIX USEr NETwork USENET (also referred to as NetNews) started in 1979 as a messaging system on top of UUCPNET, later it migrated to the Internet. Currently USENET is a collection of over 50,000 news groups where users can post and read news messages. See also UUCPNET. *Page 9*

UUCPNET . . . . . UNIX-to-UNIX Copy Protocol NETwork UUCPNET was a point-to-point overlay network on top of the telephone network. UNIX hosts would call up other UNIX hosts for information exchange. *Page 14*

VLR . . . . . Visitors Location Register The VLR is a temporary database of the subscribers who have roamed into the particular area which it serves. *Page 27*

VoIP . . . . . Voice over Internet Protocol VoIP allows transmission of telephone calls, in the form of small digitized voice packages, over the Internet. VoIP is also referred to as Internet telephony, IP telephony, and VOI. *Page 7*

WAN . . . . . Wide Area Network A WAN is a communication network that covers a broad area like a country. The largest WAN is the Internet, covering most of the world. Compared to a LAN connection speeds are usually low and communication costs are usually high. See also LAN. *Page 139*

WAP . . . . . Wireless Application Protocol WAP allows users to access information via handheld wireless devices such as mobile phones, pagers, two-way radios, smartphones and communicators. See also cHTML. *Page 8*

WhatsApp . . . . . WhatsApp Messenger WhatsApp is a messaging application for mobile Internet devices often favored over SMS and other non Internet messaging systems due to their payment model. WhatsApp can relay text, images, video and audio messages. *Page 8*

WML . . . . . Wireless Markup Language WML is a, HTML like, content description language for WAP, describing several screens (named cards) per file. This in contrast to cHTML, which can describe only one screen per file. See also WAP and cHTML.

WWW . . . . . World Wide Web WWW is a system of servers that support HTML documents. See also HTML. *Page 12*

XML . . . . . eXtensible Markup Language XML is a specification language allowing the creation of customized markup tags, enabling the validation, and interpretation of data between different applications.

XMPP . . . . . eXtensible Messaging and Presence Protocol XMPP is an open, XML-based protocol for server-to-server IM and presence. XMPP is often identified with the Jabber program, the best known, but not the only, IM application that uses XMPP. See also IM. *Page 19*

Bibliography

[1] A. Agarwal. Retrospective: The MIT Alewife Machine: architecture and performance. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 103--110, New York, NY, USA, 1998. ACM.

[2] G.A. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] T.T. Ahonen. Tomi Ahonen Almanac 2010. Mobile Telecoms Industry Review. `http://www.tomiahonen.com/ebook/almanac.html`, 2010.

[4] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. In *SOSP '75: Proc. of the fifth ACM symposium on Operating systems principles*, pages 67--74, New York, NY, USA, 1975. ACM.

[5] P. Albitz and C. Liu. *DNS and BIND*. O'Reilly & Associates, Sebastopol, CA, 3rd edition, 1998.

[6] G. Appenzeller et al. The Mobile People Architecture. *ACM Mobile Comp. and Commun. Rev.*, 1(2), 1999.

[7] S. Arbanowski and S. van der Meer. Service personalization for unified messaging systems. In *Comp. and Commun., 1999. Proc. IEEE Int. Symposium on*, pages 156--163, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[8] M. Astley, D.C. Sturman, and G.A. Agha. Customizable middleware for modular distributed software. *Commun. ACM*, 44(5):99--107, 2001.

[9] D. Atkins, W. Stallings, and P. Zimmermann. PGP Message Exchange Formats. RFC 1991 (Informational), August 1996. See also RFC 4880.

[10] D. Barber. GlobalCom: a unified messaging system using synchronous and asynchronous forms. In *PPPJ/IRE*, pages 141--144, 2002.

[11] S. Barber. Common NNTP Extensions. RFC 2980 (Informational), October 2000. Updated by RFCs 3977, 4643, 4644.

[12] S. Belville. *Zephyr on Athena*. MIT Press, Cambridge, MA, 1990.

[13] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, and Y. Tian. *A comparison of join algorithms for log processing in MapReduce*. In *Proc. Int. Conf. on Management of Data*, volume SIGMOD 2010, pages 975--986, New York, NY, USA, 2010. ACM.

[14] R. Blood. *The Weblog Handbook*. Perseus Publishing, Cambridge, MA, 2002.

[15] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP Message Format. RFC 4880 (Proposed Standard), November 2007. Updated by RFC 5581.

[16] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[17] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. Comp. Syst.*, 190(3):332--383, 2001.

[18] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[19] D.L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84--90, 1981.

[20] L. Chong, S. Cheung Hui, and C. Kiat Yeo. Towards A Unified Messaging Environment Over The Internet. *Cybernetics and Systems*, 30:533--549, 1999.

[21] Cognitive Science Laboratory Princeton University. *WordNet: a lexical database for the English language*. `http://wordnet.princeton.edu/`, January 2000.

[22] D.E. Comer. *The Internet Book*. Prentice Hall, Upper Saddle River, N.J., 3rd edition, 2000.

[23] M. Crispin. Internet Message Access Protocol - Version 4rev1. RFC 2060 (Proposed Standard), December 1996. See also RFC 3501.

[24] M. Day, S. Aggarwal, G. Mohr, and J. Vincent. Instant Messaging / Presence Protocol Requirements. RFC 2779 (Informational), February 2000.

[25] M. Day, J. Rosenberg, and H. Sugano. A Model for Presence and Instant Messaging. RFC 2778 (Informational), February 2000.

[26] R. Dreher and L. Harte. *Signaling System 7 Basics*. APDG Publishing, Fuquay Varina, NC, 2nd edition, 2002.

[27] P.Th. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. Epidemic Information Dissemination in Distributed Systems. *IEEE Computer*, 37(5):60--67, 2004.

[28] R. Gellens, C. Newman, and L. Lundblade. POP3 Extension Mechanism. RFC 2449 (Proposed Standard), November 1998. Updated by RFC 5034.

[29] K. Gerwig. Business: the 8th layer: e-mail outsourcing sends a message. *netWorker*, 3(2):13--16, 1999.

[30] P. Healy, D. Barber, and B. Nolan. Developing unified messaging system apps in JAVA. In *PPPJ '03: Proc. of the 2nd Int. Conf. on Principles and Practice of Programming in Java*, pages 137--138, New York, NY, USA, 2003. Computer Science Press, Inc.

[31] M.R. Horton and R. Adams. Standard for interchange of USENET messages. RFC 1036, December 1987. See also RFCs 5536, 5537.

[32] V. Jacobson. *If a Clean Slate is the solution what was the problem?* Stanford *Clean Slate* seminar, February 2006. `http://cleanslate.stanford.edu/seminars/jacobson.pdf`.

[33] M. Jelasity, S. Voulgaris, R. Guerraoui, A.M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comp. Syst.*, 25(3), August 2007.

[34] R. Jennings. *Cost of Spam is Flattening. Our 2009 predictions.* `http://ferris.com/2009/01/28/cost-of-spam-is-flattening-our-2009-predictions/`, January 2009.

[35] C. Kalt. Internet Relay Chat: Architecture. RFC 2810 (Informational), April 2000.

[36] C. Kalt. Internet Relay Chat: Server Protocol. RFC 2813 (Informational), April 2000.

[37] B. Kantor and P. Lapsley. Network News Transfer Protocol. RFC 977 (Proposed Standard), February 1986. See also RFC 3977.

[38] G. Kessler and S. Shepard. A Primer On Internet and TCP/IP Tools. RFC 1739 (Informational), December 1994. See also RFC 2151.

[39] G. Kiczales, J. Lamping, A. Menhdhekar, Ch. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. European Conf. on Object-Oriented Programming*, volume 1241, pages 220--242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[40] G. Kiczales and M. Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proc. of the 27th Int. Conf. on Software eng.*, pages 49--58, New York, NY, USA, 2005. ACM Press.

[41] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001. See also RFC 5321, updated by RFC 5336.

[42] E. Klintskog, Z. El Banna, and P. Brand. A generic middleware for intra-language transparent distribution. Technical Report T2003:01, Swedish Institute of Computer Science, January 2003.

[43] F. Kon, F. Costa, G. Blair, and R.H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33--38, 2002.

[44] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Proc. of the 1st Workshop on Economics of Peer-to-Peer Systems*. in-house, 2003.

[45] M. Marx and Ch. Schmandt. CLUES: dynamic personalized message filtering. In *CSCW '96: Proc. of the ACM Conf. on Computer supported cooperative work*, pages 113--121, New York, NY, USA, 1996. ACM.

[46] D.S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, Hewlett Packard Laboratories, Palo Alto, CA, March 2002.

[47] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[48] J. Oikarinen and D. Reed. Internet Relay Chat Protocol. RFC 1459 (Experimental), May 1993. Updated by RFCs 2810, 2811, 2812, 2813.

[49] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer-Verlag, Berlin, Heidelberg, and New York, third edition, 2011.

[50] J.A. Patel and I. Gupta. Overhaul. In *Web Content Caching and Distribution*, volume 3293 of *Lecture Notes in Computer Science*, pages 34--43. Springer Berlin / Heidelberg, 2004.

[51] Pingdom. *Internet 2011 in numbers*. posted January 17th. `http://royal.pingdom.com/2012/01/17/internet-2011-in-numbers/`, 2012.

[52] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[53] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277--288, November 1984.

[54] S. Saroiu, P.K. Gummadi, and S. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *SPIE Multimedia Computing and Networking (MMCN2002)*, 2002.

[55] B. Schneier. *Applied Cryptography*. John Wiley, New York, 2nd edition, 1996.

[56] R. Sexton. *Wikipedia entry for B1FF*. `http://en.wikipedia.org/wiki/B1FF`, Unknown 2000.

[57] K. Singh and H. Schulzrinne. Unified messaging using SIP and RTSP. *IP Telecom Services Workshop*, page 7, september 2000.

[58] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Replication for web hosting systems. *ACM Comp. Surveys*, 36(3):291--334, 2004.

[59] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[60] W.D. Tajibnapis. The design of a topology information maintenance scheme for a distributed computer network. In *ACM 74: Proc. of the 1974 Ann. Conf.*, pages 358--364, New York, NY, USA, 1974. ACM Press.

[61] A.S. Tanenbaum and R. van Renesse. A Critique of the Remote Procedure Call Paradigm. In *Proc. of the EUTECO 88 Conf.*, pages 775--783, Vienna, Austria, November 1988. North-Holland.

[62] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, September 2003.

[63] S. Vinoski. A Time for Reflection. *IEEE Internet Comp.*, 9(1):86--89, 2005.

[64] S. Voulgaris, D. Gavidia, and M. van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13:197--217, 2005.

[65] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Mountain View, CA, November 1994.

[66] J.M.S. Wams and M. van Steen. A Flexible Middleware Layer for User-to-User Messaging. In *Distributed Applications and Interoperable Systems*, volume 2893 of *Lecture Notes in Computer Science*, pages 297--309. Springer Berlin / Heidelberg, 2003.

[67] J.M.S. Wams and M. van Steen. Unifying User-to-User Messaging Systems. *IEEE Internet Comp.*, 8(2):76--82, 2004.

[68] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptolog -- EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 561--561. Springer Berlin / Heidelberg, 2005.

[69] A. Westine and J. Postel. Problems with the maintenance of large mailing lists. RFC 1211 (Informational), March 1991.

[70] R. Wieringa and W. de Jonge. Object Identifiers, Keys, and Surrogates--Object Identifiers Revisited. *Theory and Practice of Object Systems*, 1(2):101--114, 1995.

[71] D.R. Woolley. PLATO: The Emergence of Online Community. *Matrix News*, 1994.

[72] Małgorzata Wrzesińska. A Secure Instant Messaging System. `http://students.mimuw.edu.pl/SR/prace-mgr/wrzesinska/thesis6.html`, June 2002.