View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at VU

Unifying Internet Services Using Distributed Shared Objects

Philip Homburg
Maarten van Steen
Andrew S. Tanenbaum

Internal report IR-409
October 11, 1996

Abstract

Developing wide area applications such as those for sharing data across the Internet is unnecessarily difficult. The main problem is the widespread use of a communication paradigm that is too low level. We will show how wide area application development can be made easier when using distributed shared objects instead of a communication-oriented model. An object in our model is physically distributed, with multiple copies of its state on different machines. All implementation aspects such as replication, distribution, and migration of state, are hidden from users through an object's interface. In this paper, we concentrate on the application of distributed shared objects, by providing an outline of a middleware solution that permits integration of the Internet services for e-mail, News, file transfer, and Web documents.

vrije Universiteit

1 Introduction

Constructing wide area applications, such as those for sharing data across the Internet, often requires a substantial development effort. This is mainly caused by the lack of proper communication facilities as offered by the underlying operating systems and middleware solutions. For example, most operating systems provide only a very simple socket-based network interface. On the other hand, middleware solutions such as DCE [13] or CORBA [11] offer more advanced communication facilities such as RPC or remote object invocation, but lack support for related issues such as replication. Also, it is yet unclear whether these systems can scale to millions of users and billions of objects.

To illustrate the problems for sharing data in wide area networks caused by the lack of proper communication facilities, we focus in this paper on four applications used in the Internet, and their corresponding protocols. These are electronic mail (SMTP [12]), USENET News (NNTP [9]), file transfer (FTP [3]), and the Web (HTTP [2]). The protocols have the following in common:

- Each protocol is designed to support interoperatability between different implementations of that protocol.
- Each protocol handles the transfer of data between sites, but ignores aspects related to, for example, data management, replication, and security.
- They are all designed as a relatively simple layer directly on top of the TCP/IP protocol stack.

Accompanying a new application with its own transfer protocol has generally been the approach followed in the Internet. This approach has two major drawbacks. First, enhancements to a protocol are hard to realize as it requires worldwide consensus as well as changing many independently written implementations. Second, although very different implementations of the *same* protocol can easily interoperate, interoperatability between *different* protocols has shown to be hard. In order to easily adapt and integrate service, it is essential that the level of abstraction of network communication facilities offered by current systems is raised. In particular, we advocate that the underlying system should provide an object-based model in which distributed objects are used as the basis for sharing and exchanging information. How this approach can integrate applications and simplify software design is the main topic of this paper.

The paper is organized as follows. In Section 2 we take a closer look at the functionality of the four Internet services to see what that have in common. In Section 3 we explain our model of distributed shared objects. Integration of the four services in terms of our object model is discussed in Section 4, while the integration with existing implementations is presented in Section 5. Our conclusions are stated in Section 6.

2 The Four Internet Services

We start with taking a closer look at each of the four Internet services. Figure 1 shows how the protocols differ, where we distinguish naming, data distribution, replication of data, grouping of documents, authentication and secrecy, self-containedness, and data encoding. These differences are discussed below.

2.1 Common functionality

Naming. The Mail system names *users* at *domains*. Although a mailbox is globally unique, the interpretation of the user name is left to individual mail transfer agents (MTA). SMTP only defines a map-

	Mail	News	FTP	WWW
Naming	Mailbox (user@domain)	Newsgroup	Host + path	URL
Distribution	Push	Push	Pull	Pull
Replication	None	Flooding	Caching + DNS tricks	Caching + DNS tricks
Containers	Mailbox	Newsgroup	Directories	HTML + Frames
Authentication	PEM or PGP	None (PGP)	Username + Password	Username + Password
Secrecy	PEM or PGP	None	None	None
Complete	No	Theory: Yes Practice: No	Theory: Yes Practice: No	No
Encoding	7-bit + MIME + 8-bit extensions	7-bit + MIME	7-bit text + 8-bit binary (user has to guess)	8-bit + content type

Figure 1: Features of four Internet protocols.

ping from the domain part of a mailbox to the machine(s) on which the MTA runs that accepts mail for that domain. The News system names *newsgroups*. Newsgroup names are valid in a particular domain ("distribution"). Names for local newsgroups are typically not unique, which causes confusion if an article is cross-posted in several domains. Articles in a single newsgroup are uniquely distinguished by the their message identifier. Naming in FTP is done for *files*, and the *host* of the file system. The only naming convention adopted is that of the FTP server, whose name is coupled to the DNS host name on which the server is running. Finally, the World Wide Web uses *Uniform Resource Locators* (URLs) for naming Web pages and other documents. The distinguishing feature is that a protocol identifier makes part of a URL, allowing documents available under HTTP, FTP, Mail, and News, to be named within a single naming scheme.

Data Distribution. Both Mail and News actively *push* data towards the destination, which is a mailbox, or a peer news server, respectively. On the other hand, for FTP and Web documents, the user is expected to explicitly *pull* the document from a server or a cache to its own location.

Data Replication. The only system for which replication is defined is News. News articles are replicated using a flooding protocol. Mail messages are not replicated. If a message has multiple recipients, copies are sent to each recipient separately. FTP and HTTP do not support replication, although some replication is done manually. Using mirror sites, entire FTP and HTTP servers can be replicated, and registered under a single DNS name using round-robin DNS. Caches can be seen as a weak form of replication. Copies of the content of Web documents are stored in proxy Web servers. However, note that standardized cache consistency protocols are lacking entirely.

Document Containers. The four applications differ in how documents are collected and organized. Mail messages are collected in mailboxes; News articles are collected in newsgroups; FTP supports the traditional concept of hierarchically organized directories. In contrast to Mail and News, FTP provides the user with full support for manipulating directories. The World Wide Web does not support

document containers at all. It is only possible to have references to other documents contained in a Web page, but there is no support for collecting groups of pages into something analogous to a directory.

Authentication and Secrecy. Security is primarily supported only in Mail and News. PEM (Privacy-Enhanced Mail) is a standard protocol for authentication and secrecy of mail messages. In practice, PGP (Pretty Good Privacy) is commonly used to authenticate both Mail messages and News articles, and to encrypt Mail messages. Authentication for FTP and HTTP is limited to simple user name/password combinations. Nonstandard security extensions exist for Web documents, which are specific to Web browsers, such as SSL for Netscape™.

Completeness. None of the four services is really self-contained, with the possible exception of the News system. NNTP supports both posting and reading News articles. Furthermore, News articles may contain control fields to create and delete newsgroups. Unfortunately, the usefulness is limited by the lack of authentication of News articles. There is no standard protocol for introducing a new News server to neighboring servers. Initially, the Mail system only defined a protocol for *sending* mail to a mailbox, but lacked a protocol for *reading* mail. POP (Post Office Protocol) was introduced to solve this problem. However, there are no standard protocols for creating or deleting mailboxes. FTP provides a fairly complete set of commands for maintaining a file system. For various reasons these commands are typically not used, and instead the underlying local file system is modified directly. Finally, HTTP is also not self-contained. For example, it relies on existing file systems and has only limited commands for adding new documents to an existing directory.

Data Encoding. Data encoding also differs between the various protocols. Mail and News allow 7-bit ASCII data transfers. MIME extends this by defining standard encodings for binary and 8-bit text. Extensions to SMTP allow passing 8-bit data directly from an SMTP client to an SMTP server if they both implement the extension. In practice, almost all Mail and News transport agents allow 8-bit text encodings such as Latin-1. FTP supports both text files and data files. Unfortunately, the user has to guess the correct type, and explicitly configure either a binary or ASCII transfer. HTTP supports 8-bit binary data, and provides the recipient with a content type that describes the data.

2.2 Unifying the Different Services

Some differences between the four services are historical, but others are caused by different underlying *communication models*. The communication model underlying Mail is that of message exchange between two parties. The News model of communication is very similar, except that message exchange takes place within a group of participants. However, file transfer is radically different. It is based on files organized in a file system at a particular site. Operations mainly deal with manipulating directories and explicitly copying files between a remote and a local file system. Likewise, the model underlying the Web is also different. The Web model assumes a world of Web pages, each page located at a particular site. Each page can refer to other pages through hyperlinks. When a client activates a hyperlink the associated Web page is copied to the client's site so that it can be read. However, these models are only implicit: all four services are described in terms of *transfer* protocols. And indeed, this makes the service incomplete and often highly inflexible. For example, data management is lacking and where applicable, replication strategies are hard-wired into a protocol.

A solution towards integration adopted in the Web community, is to provide an single Web browser that supports each protocol. However, providing an integration at the level of the user interface solves

only a very small part of the problem. In particular, the distinction between how the shared data is accessed and manipulated remains: there is no integration of underlying communication models. New programs that need to access multiple kinds of data (e.g., Mail messages and Web pages) must still deal with each one separately.

Rather than integrating the services directly, or choosing ad-hoc solutions such as adopted in browsers, we advocate that the underlying systems on which data sharing services in large-scale internets are to be based, should be adapted first. In this paper, we show that an approach based on the concept of distributed shared objects can alleviate many of the problems encountered with integrating such services.

3 Distributed Shared Objects

In this section we briefly describe our model of distributed shared objects. We first describe what a distributed shared object is, and then focus on the support for replication.

3.1 Object Organization

In our model, processes interact and communicate through **distributed shared objects** [7, 17, 8]. Each object offers one or more interfaces, each consisting of a set of methods. Objects are passive, but multiple processes may access the same object simultaneously, allowing them to communicate by reading and changing the object's state through method invocations. Changes to the object's state made by one process are visible to the others. A major distinction with other object-based models is that objects are physically distributed, meaning that active copies of an object's state can and do reside on multiple machines at the same time. However, processes are not aware of this: state and operations on that state are completely encapsulated by the object. This means that all implementation aspects, including communication protocols, replication strategies, and distribution and migration of state, are part of the object but are hidden behind its interface. This model is illustrated in Figure 2.

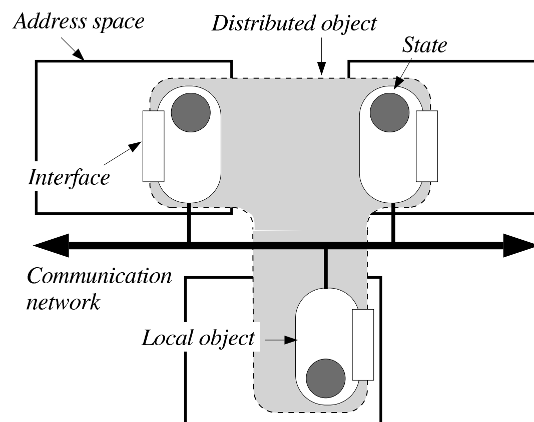

Figure 2: Example of a distributed object spanning multiple address spaces.

Distributed objects are built from local objects. A **local object** is confined to a single address space. It can contain both code and data, although code is typically stored in a **class object**. This is a local object which contains the method implementations for objects belonging to the same class. An object interface, of which there may be several per object, is implemented as a separate table that stores pointers to the state and methods of the object. There are standard interfaces for objects as well as

class objects that always have to be provided. Local objects can be combined into **composite objects**. A composite object exports interfaces that can contain methods from the various constituent objects. Our model allows for dynamically loading class objects by a running program. Distributed objects consist of local objects, generally placed in different address spaces. The local objects are responsible for establishing communication between the different address spaces in such a way that a user of the distributed object is provided with a consistent view on the object's state. Local objects are thus responsible for hiding distribution, replication, and migration of the object's state from the user.

Before a process can invoke an object's method, it has to **bind** to the object. Binding involves installing and initializing one or more local objects in the process' address space. These local objects offer an implementation of the interface used by the process. In contrast to many other object models which essentially follow the client/server approach, we support a *symmetric* model of computation: once a process is bound to an object, it forms part of the object's overall implementation. We therefore say that a process *participates* in the implementation of a distributed object. There is no client and no server.

Figure 3: Implementation of a distributed object.

Each local object forming part of a distributed object is a self-contained composite object. This is shown in Figure 3. A **communication object** encapsulates the implementation of communication channels connecting the address space to remote parts of the distributed object. It offers a relatively simple communication interface to a **replication object** which is responsible for keeping the possible replicas of local state consistent. The local states jointly constitute the distributed object's state. The **semantics object** contains the implementation of the actual functionality of the distributed object, and thus determines the *type* of the object. The **control object**, finally, coordinates access to the semantics object. It regulates method invocations from the process that is bound to the object, as well as invocations from pop-up threads activated on account of communication between different address spaces. We expect to automatically generate control objects, similar to stub generation in RPC systems.

The composite object exports two interfaces to the user of the distributed object: the user-defined interface provided by the control object and a distributed object interface. Internal to the composition are the communication, replication, control, and receive interfaces.

3.2 Replication

To be able to use different replication and communication objects with a single semantics object and derived control object, we need standard interfaces between the control object and replication object, and between the replication object and the communication object. The interface to the communication object is straightforward. There are methods to set up a connection, to join and to leave multicast groups, and to send data. Incoming data and connect indications are handled by pop-up threads, which invoke methods on callback interfaces exported by the replication object.

Replication Interface	
Init	Provide the replication object with the control interface
Start	Start execution of a method invocation
Send	Send method arguments to a remote address space
Blocked	Method invocation blocked
Finish	Method invocation is completed
Request_done	Previously blocked operation is completed

Control Interface	
Handle_request	Invoke an incoming operation on the semantics object
Cancel_request	Cancel a blocked operation
Retry	Retry blocked operation

Figure 4: Replication and control interface.

Figure 4 lists the methods of the replication interface exported by the replication object and the callback interface of the control object. During initialization, the control object invokes the init method of the replication object to pass a reference to its (callback) control interface. Execution of a method consists of three steps. This is shown in Figure 5. First, the control object invokes the start method on the replication object. This gives the replication object the opportunity to synchronize with other replicas, acquire locks, etc. The second step is split into three different variants: remote execution, local execution and active replication. The start method returns a value specifying which of the three variants is required.

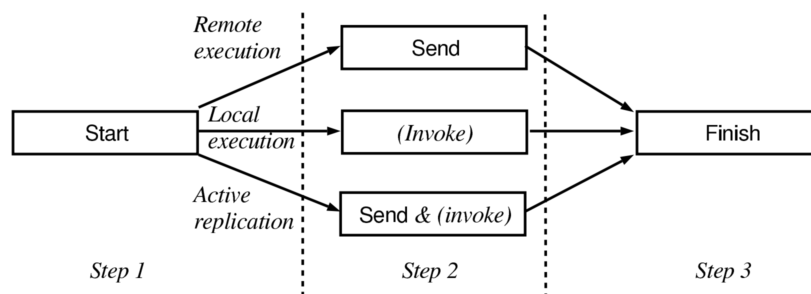

Figure 5: A method invocation.

For remote execution, the control object marshals the arguments of the method invocation, and invokes the send method. The replication object forwards the arguments to one or more remote address spaces, waits for marshaled results, and returns those results to the control object. For local execution, the control object simply invokes the appropriate method on the semantics object.

Active replication combines both remote execution and local execution. First the control object marshals the arguments and invokes the send method. The arguments are sent to all address spaces with a replica of the object's state. The replication objects in different address spaces order the method invocations to maintain proper state consistency. When the moment has come to actually execute the method, the replication object returns from the send method and the control object invokes the operation on the semantics object.

Finally, the control object invokes the finish method to give the replication object the opportunity to release locks and distribute the new state of the object.

For incoming calls, the replication object invokes the handle_request method on the control object. The blocked, request_done, cancel_request, and retry methods deal with guarded operations. When a method invocation of the semantics object blocks, the control object invokes the blocked method on the replication object. The control object creates continuations for blocked invocations and puts them on a list. The replication object can call cancel_request to abort blocked operations. The retry method tells the control object to retry blocked method invocations. And when a blocked invocation has completed, the control object invokes request_done to pass the marshaled results to the replication object.

3.3 Naming Distributed Objects

Distributed objects are generally registered with a *directory service*. This service maps one or more user chosen names for an object to a set of contact addresses. These contact addresses allow a newly created local object to setup a communication channel to the distributed object. When the communication channel is established, the new local object is said to participate in the distributed object.

We implement the directory service as two independent services: the name service and the location service. The name service provides a worldwide name space that maps object names to **object handles**. An object handle uniquely identifies an object and is mapped by the location service to a set of contact addresses (see [16] for further details). This two-layer approach allows an object to change its set of contact addresses without affecting the mapping at the name service. An object's set of contact addresses changes when, for example, the object migrates, or when it wants to provide access to additional replicas of the state of the object.

The distributed object interface is a standard interface that is used during binding to inform the new local object about the distributed object's set of contact addresses as returned by the location service.

3.4 Security

A detailed explanation of our security model is beyond the scope of this paper. In our model, authentication and access checks are done when a new local object tries to connect to a contact point of a distributed object. Distributed objects contain access control lists that specify which principals are allowed to invoke methods on the object. Principals can represent individual users, users in a certain role, or groups of principals. The communication channels set up during binding are secure, that is data sent over the communication channel are encrypted and signed.

4 Mail, News, FTP, and WWW as Distributed Shared Objects

In this section we show how distributed shared objects can be used to unify the four Internet services from Section 1. The interfaces we describe below are based on a prototype system we are currently developing.

4.1 Main Interfaces to Documents and Containers

Distributed shared objects provide us with the means to properly integrate the four services we consider in this paper. The integration itself is based on an approach in which we distinguish the concept of a *document*, and that of a *container*. We describe a uniform **document object** which is suitable for containing information that is now encoded into separate documents for Mail, News, FTP, and WWW, respectively. Document objects are collected into **container objects**. A container object contains references to document objects, but also to other container objects allowing for a graph-based organization.

We use three interfaces to access document and container objects. The main interface of a document object is like a simple file interface with read and write methods. Container objects provide a standard naming interface with list, lookup, add and delete methods. Finally, we have a meta-data interface which provides access to *(attribute,value)* pairs. Figure 6 lists the methods of the naming and meta-data interface.

Naming Interface	
List	Return a list of directory entries
Lookup	Lookup a directory entry, return the object handle
Add	Add a directory entry
Delete	Delete a directory entry

Meta-data Interface	
Setattr	Set a (name, value) pair
Getattr	Lookup a name
Listattrs	Return a list of attributed

Figure 6: Main interfaces to document and container objects

It is important to note that in order to construct applications using our object model, a developer should first concentrate on the design of object interfaces. The next step is to develop a semantics object that implements the functionality of the interface. However, different implementations of the same interface may co-exist. When it is known to a process which interface it requires, it can select an appropriate implementation and dynamically load that implementation into its address space. In our example, both the naming and the meta-data interface can be implemented as an associative array. The file interface can be simply implemented as an array of bytes.

Document and container objects can be used to implement our four example Internet services as follows. Mail messages and News articles are implemented as document objects, with the meta-data interface being used to manage header information (such as date, subject, sender, MIME-version, content-type, etc.). Mapping mailboxes and newsgroups to container objects is somewhat harder, as messages and articles are typically not named. There are two solutions to this problem: (1) messages and articles are entered under an arbitrary name, for example their ID, or (2) the implementation of a container object can ignore the name argument altogether.

Files and directories on an FTP server map quite easily to document objects with a file interface, and container objects with a directory interface, respectively.

The World Wide Web consists of pages that are linked together through hyperlinks, and there is no concept of a directory. There are two ways to map Web pages. The first one is to store a Web page in a document object, use a container object to structure the name space, and treat hyperlinks as symbolic links. This means that the name of the file containing the Web page is used as a link. This corresponds with current practice. A disadvantage of symbolic links is that the target of the link can be deleted

or renamed without updating the link. The alternative is to implement a Web page as an object with both a file and a directory interface. The directory can contain "hard" links to related documents, and the document itself contains references to its directory entries. In our model, object handles are the equivalent to hard links, and we use them to refer to objects. An object handle does not change when its associated object is renamed.

4.2 Replication

To implement directory and document objects for the four services we need a collection of replication objects. Replication objects are kept in libraries. When an object is constructed, the replication strategy is determined by the choice of replication object selected.

Client/Server Replication. The simplest replication object occurs in client/server computing where the entire state is in a single server and no replication is provided. Note that in some cases, implementations in terms of distributed shared objects of existing applications like search engines and database frontends, can only be done through client/server replication. This is also the case, for example, with functionality implemented through CGI scripts.

Primary-Backup Replication. A simple extension to the client/server replication object is primary-backup replication [4]. The state of the distributed object is replicated and one copy is designated (or elected) as the primary copy. All operations on the distributed object are forwarded to the primary. The primary executes the method and updates all other backup copies before sending a reply to the requesting address space. This scheme provides increased fault-tolerance and availability over client/server replication.

Replication of Immutable Objects. In the Mail and News systems, documents (messages and articles) are immutable: they do not change after they are sent or posted. Replication of immutable objects is trivial: each address space that needs to access the object gets a copy of its state. Since the object cannot be changed, there is no need for a data consistency protocol.

Active Replication. To be able to support mutable objects with a large number of replicas, we can use active replication where changes to the object are either flooded, or multicast using a spanning tree. In this scheme, each replication object maintains a network connection to a few neighboring replication objects. Local changes to the object's state are then also forwarded to its neighbors.

Push/Pull Replication. To improve the performance of accessing WWW and FTP documents, we introduce push/pull replication. Push/pull replication combines a replication scheme for a small number of copies (such as primary-backup replication) with caches. Changes to the state of the object are actively pushed to a relatively small number of replicas. Processes that use the distributed object fetch a copy of the state of the object through cache servers. Read-only caches maintain their copy of the state up-to-date by discarding copies after some time, and by polling authoritative replicas of the object.

4.3 Example

To illustrate our approach, consider the following example in which a distributed object uses primary-backup replication. Figure 7 shows the flow of control when invoking one of the object's methods.

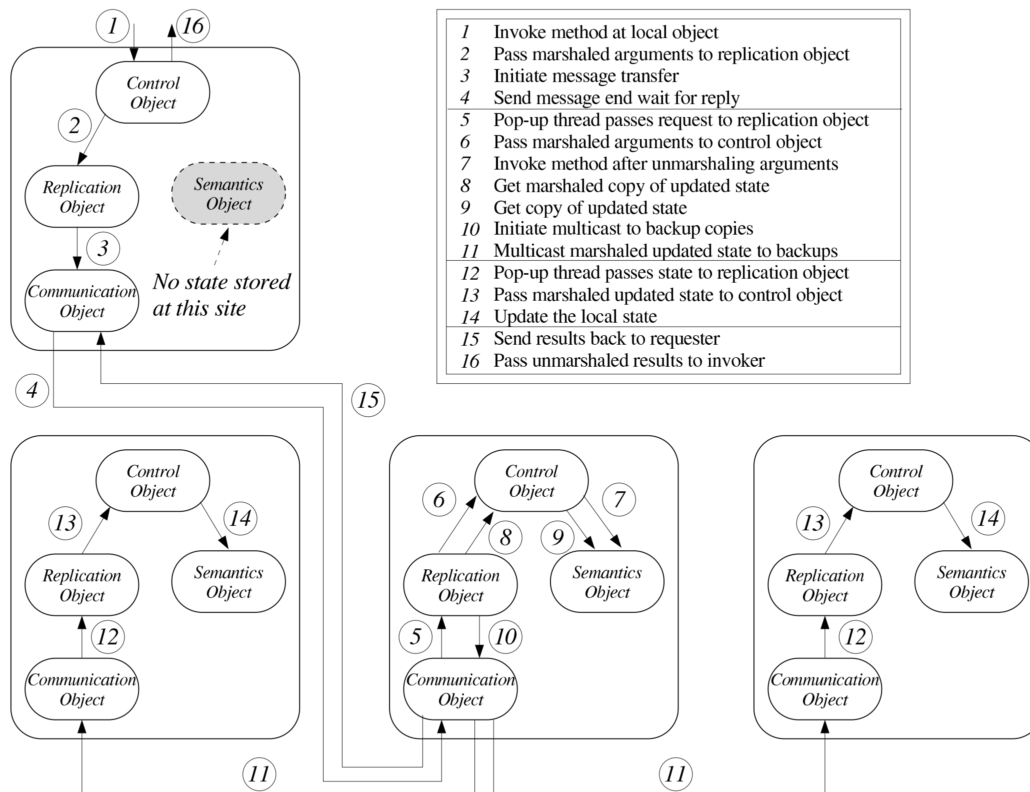

Figure 7: A method invocation on a distributed object using primary-backup replication.

The method invocation starts at the control object in an address space without a local replica (1). The control object passes the marshaled arguments to the replication object (2), which in turn calls the communication object to send a request to the primary copy and wait for a reply (3 and 4). When the request arrives at the communication object of the primary copy, the communication object creates a pop-up thread and passes the contents of the request to the replication object (5). The replication object passes the marshaled arguments to the control object (6), which invokes the semantics object (7). The semantics object returns the results of the method invocation to the control object and the control object marshals the results and returns them to the replication object.

There are three ways to propagate the changes to the other replicas: the primary can either (1) send a new copy of the state, (2) send the changes in the marshaled version of the state, or (3) send the marshaled arguments and expect the other replicas to execute the method themselves. The last alternative is close to active replication. Figure 7 shows the first alternative, where the replication object calls the control object to get a marshaled copy of the new state of the semantics object (8), the control object in turn calls the semantics object (9), then marshals the result and returns it to the replication object.

Next, the replication object calls the communication object (10) to send the new state to the other replicas (11). The communication objects for the other replicas create pop-up threads and invoke their replication objects (12), which then pass the marshaled state to the control objects (13). The control objects install the state in the semantics objects (14). After the new state is successfully shipped, the replication object of the primary copy returns the marshaled results to the communication object, which sends them back to the address space that started the method invocation (15).

Finally, the communication object in the originating address spaces returns the reply packet, which

is decoded by the control object, and the results of the method invocation are returned to the caller (16).

The scheme described above can be enhanced by allowing requests to be sent to a backup copy in addition to the primary copy. If the request modifies the state of the object, the backup has to forward the request to the primary copy. However, if the request is a read operation, it can be executed locally. This increases the throughput and decreases the latency for read operations. The other replication strategies work in an analogous way.

4.4 Replication in the example services

The replication objects described above can be used to match the replication strategies embedded in the four example Internet transfer protocols. In most cases, only client/server replication is needed to describe the present situation. For example, mailboxes and Web documents hardly make use of replication, and only relatively few FTP sites are mirrored. Newsgroups, on the other hand, use active replication through flooding.

Each service can be improved by using more advanced replication objects. As Mail messages can be expected to immutable, mailboxes are obvious candidates for primary-backup replication to provide increased fault-tolerance. FTP and Web applications can profit from replication objects implementing push/pull replication. However, other combinations are also possible. Highly popular Web pages, for example popular manual pages for a particular software product, might be flooded to all sites using that product. Similarly, small, popular newsgroups might use push/pull replication as well. Finally, it is also possible to mail or post documents that are not immutable, but use active replication or push/pull replication instead.

It is important to note that improving each service does not affect the functionality of that service, nor is it necessary to adapt the communication protocol on which the service depends. Instead, each improvement can be made simply by replacing the replication strategy through a different replication object.

5 Evolution path

In this section we describe how to integrate the current protocols with our solution based on distributed shared objects. Most of the interaction in the current protocols are client/server-oriented, therefore we have to deal with two situations per protocol: an existing client using a distributed object, and a new client using an existing server. Figure 8 shows a standard way to solve this problem using *proxies* [14].

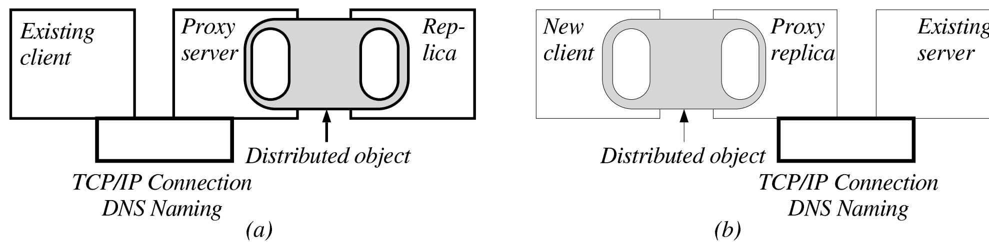

Figure 8: Existing clients/servers and distributed objects.

Figure 8(a) shows an existing client connected to a proxy server that has bound to a distributed object. Figure 8(b) shows a client using a distributed object implemented in such a way that operations on the object are forwarded by the proxy to an existing server. A proxy has to implement two

functions. First, it has to convert requests arriving over a TCP/IP connection to method invocations on a distributed object and vice versa. A proxy also has to provide a mapping between the different naming systems. Note that in the second configuration, when a new client uses an existing server, it is possible to omit the proxy replica and let the client process communicate with the server directly. This is possible because object implementations are loaded dynamically, and because objects completely encapsulate all aspects of communication. Below we discuss proxies for the four services.

Mail. A proxy server for e-mail has to implement SMTP to accept new mail and POP to allow reading of mail. The proxy server creates a document object from an incoming mail message and stores the object in a mailbox container object. The fields in the header of a mail message are stored as meta data in the document object. Similarly, the proxy returns directory listings of a mailbox object and the contents of document objects in response to POP requests.

Conversely, when a distributed object provides access to an existing mailbox, the proxy has to fake one container object and multiple document objects per mailbox. Requests to add new documents to the directory are forwarded over SMTP to the real server. Lookup request for contents of the container and the contents of documents are implemented using POP.

Name translation is the most interesting aspect of the proxies. The server proxy has to create old-style (i.e. "user@domain") e-mail addresses that correspond uniquely to object names. The domain part is typically fixed for a particular proxy, there are however several alternatives for the user part. One approach is to use the object name for the user part. For example, a mailbox known as a distributed object named /org/foo/bar/mailbox might be renamed to /org/foo/bar/mailbox@proxy.com. Another alternative is to export all mailboxes that are listed in one directory. In this case /org/foo/mailboxes/bar might be converted to bar@proxy.com. Yet another alternative is to store an explicit mapping in the proxy server.

Providing object names for existing mailboxes can be done automatically. All that is needed are some proxy directory objects that fake other directory objects based on DNS domains and on information returned by SMTP servers. For example, an e-mail address like philip@cs.vu.nl might get the object name /dns/nl/vu/cs/philip. The proxy directory objects would create nl/vu/cs based on information stored in DNS, and the proxy container object creates philip after querying the appropriate SMTP server for cs.vu.nl.

News. Posting and reading news is similar to e-mail. There are two major differences. First, the NNTP protocol has a command for listing available newsgroups. This requires the proxy server to maintain a list of all newsgroup container objects. This is possible due to the second difference: there are only a limited number of newsgroups so it is quite possible to explicitly store the mapping between newsgroup names and object names. In addition to posting and reading news, NNTP also implements a replication protocol: the exchange of news articles between neighboring servers. The NNTP command to support replication returns a list of articles recently added to a newsgroup.

FTP and HTTP. FTP and HTTP proxies can be written using the same techniques as e-mail proxies, with one important extension. Most WWW browsers support proxy HTTP daemons. These daemons are often used to provide site-wide caching for firewall security, but also for performance reasons [5]. A caching HTTP proxy is a good way to provide existing browsers with access to all distributed objects. This in contrast to the approaches described so far, which use a proxy for a collection of objects on a single server.

6 Discussion

The main (technical) advantage of the four protocols compared to other protocols for e-mail, news and file transfer is their simplicity. All four are implemented as a small layer on top of a TCP/IP connection. Naming is typically left to DNS. Each protocol allows for straightforward implementations that have a good chance of being interoperatable with other implementations of that same protocol.

In contrast, our system is more complex. A single object implementation is composed of four sub-objects, with multiple implementations for each of those sub-objects. The naming system names individual objects instead of machines, so that name spaces will be much larger. The main advantage of our scheme is that all aspects of communication can be hidden in distributed objects. This leads to much simpler implementations of applications and servers. So far, this approach has only be followed with Fragmented Objects [10], although that model has not been targeted towards wide area systems. To our knowledge, nearly all other object-based models assume that an object's state is not distributed, but instead is placed entirely in a single address space. This makes it much harder to adopt object-specific schemes for state replication, distribution, and migration. With our model, object-specific schemes are straightforward.

Important in our system is its support for dynamic loading of code. This allows us to deal with the diverse environments that exist in a wide area network, and to separate applications from the implementation of objects. This provides the flexibility to deploy new communication protocols and replication algorithms without affecting existing executables. However, for our approach to be fully usable, some problems still need to be addressed.

The main problem is the security risks associated with loading new code in a running executable. Experience with Java [1] shows that downloading code over the Internet is not without risk [6]. However, in our case we expect fewer risks as class implementations will generally be locally available, and maintained by the user or by the local system administrator. Unlike Java, we require conformance at the level of interfaces, not on the level of implementations. Furthermore, the most obvious candidate for downloading from a remote site is the semantics object. This object interacts with the outside world only through a control object, so it is much easier to define and implement a security policy (see also [15]).

Two other drawbacks of dynamic loading are the impact on reliability and performance. Programs that dynamically load code are typically harder to debug due to lack of debugger support, and may experience unexpected failures when new object implementations are deployed. Dynamic linking may also preclude certain optimizations like integrated layer processing.

An advantage of a system based on dynamic loading is that the flexibility provided can be used for more than just communication and replication protocols. For example, a specific implementation of a document object might send pictures over communication channels compressed with JPEG compression, but present them to an application in GIF format. All that is required is a new semantics object, or an object that encapsulates the implementation of a distributed object and implements only the data conversion routines.

The research described in this paper is partly based on a prototype system we are currently developing. Our prototype is initially aimed at devising the proper interfaces, which have been reported here. We are not aware of any related work that tries to unify the four internet protocols discussed in this paper.

References

[1] *The Java Language Environment – A White Paper*. Sun Microsystems, Mountain View, CA, Oct. 1995.

[2] T. Berners-Lee, R. Fielding, and H. Frystyk. "Hypertext Transfer Protocol – HTTP/1.0." Internet-Draft, Feb. 1996.

[3] A. Bhushan, R. Braden, W. Crowther, E. Harslem, and J. Heafner. "File Transfer Protocol." RFC 172, June 1971.

[4] N. Budhijara, K. Marzullo, F.B. Schneider, and S. Toueg. "The Primary-Backup Approach." In S. Mullender, (ed.), *Distributed Systems*, pp. 199–216. Addison-Wesley, Wokingham, 2nd edition, 1993.

[5] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J. Worrell. "A Hierarchical Internet Object Cache." Technical Report CU-CS-766-95, Department of Computer Science, University of Colorado – Boulder, Mar. 1995.

[6] D. Dean and D.S. Wallach. "Security Flaws in the HotJava Web Browser." Department of Computer Science, Princeton University, Nov. 1995.

[7] P. Homburg, L. van Doorn, M. van Steen, A. Tanenbaum, and W. de Jonge. "An Object Model for Flexible Distributed Systems." In *Proc. First ASCI Annual Conf.* , pp. 69–78, Heijen, The Netherlands, May 1995.

[8] P. Homburg, M. van Steen, and A.S. Tanenbaum. "An Architecture for A Scalable Wide Area Distributed System." In *Proc. Seventh SIGOPS European Workshop*, Connemara, Ireland, Sept. 1996. ACM. *To appear*.

[9] B. Kantor and P. Lapsley. "Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News." RFC 977, Feb. 1986.

[10] M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. "Fragmented Objects for Distributed Abstractions." In T.L. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 170–186. IEEE Computer Society Press, Los Alamitos, CA., 1994.

[11] Object Management Group. "The Common Object Request Broker: Architecture and Specification, version 1.2." Technical Report 93.12.43, OMG, Dec. 1993.

[12] J. Postel. "Simple Mail Transfer Protocol." RFC 821, Aug. 1982.

[13] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly, Sebastopol, Calif., 1992.

[14] M. Shapiro. "Structure and Encapsulation in Distributed Systems: The Proxy Principle." In *Proc. Sixth Int'l Conf. on Distributed Computing Systems*, Boston, MA, May 1986. IEEE.

[15] L. van Doorn, P. Homburg, and A.S. Tanenbaum. "Paramecium: An Extensible Object-based Kernel." In *Proc. Hot Topics on Operating Systems V*, Orca's Island, Washington, May 1995. IEEE.

[16] M. van Steen, F.J. Hauck, and A.S. Tanenbaum. "A Model for Worldwide Tracking of Distributed Objects." In *Proc. TINA '96*, Heidelberg, Germany, Sept. 1996. Eurescom. *To appear.*

[17] M. van Steen, P. Homburg, L. van Doorn, A.S. Tanenbaum, and W. de Jonge. "Towards Object-based Wide Area Distributed Systems." In L.-F. Cabrera and M. Theimer, (eds.), *Proc. Fourth Int'l Workshop on Object Orientation in Operating Systems*, pp. 224–227, Lund, Sweden, Aug. 1995. IEEE.