A Preliminary Investigation into Dynamic Distributed Workflow

Thesis by

Daniel M. Zimmerman

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

1998
(Submitted 21 May 1998)

Acknowledgements

Thank you to my advisor, Professor K. Mani Chandy, for giving me lots of help and support and displaying remarkable understanding through the various incarnations of and delays in my M.S. work. The members of my research group—Joseph Kiniry, Adam Rifkin, Paul Sivilotti, John Thornley, Eve Schooler, and Roman Ginis—have also given me useful advice and commentary, and have been generally helpful in the work leading up to this thesis.

I must also acknowledge my good friends Guillaume Lessard, Gustavo Joseph, and Brian Muzas, who have provided distraction, support and sanity checking (of both my thesis and myself, in no particular order) over the past year or two and will hopefully continue to do so in the future. Finally, thank you to my parents, my brothers, and everyone else who, while not knowing quite what I was working on, still demanded regular progress updates and thus compelled me to make regular progress.

The work which led to this thesis has been supported in part by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244, by the CISE directorate of the National Science Foundation under Problem Solving Environments grant CCR-9527130, by the Center for Research in Parallel Computing under grant NSF CCR-9120008, by Parasoft and Novell Corporation, by an NSF Graduate Research Fellowship, and by the letter "B" and the number "5". The opinions and views in this thesis are my own, and should not be taken to represent the official policies of any of the aforementioned supporters.

This thesis is available from the Caltech Computer Science Department as technical report number CS-TR-98-09.

Production Notes This thesis was typeset in LaTeX2ε on a Power Macintosh, using version 2.0 of Blue Sky Research's *Textures* package. The typeface used for the main text is Hoefler Text by the Hoefler Type Foundry, the sans serif font is Computer Modern Sans by Donald Knuth, and the typewriter font is Computer Modern Typewriter, also by Donald Knuth.

Abstract

In this thesis, we describe the concept of dynamic distributed workflow. We briefly discuss three possible approaches to the construction of a system to support dynamic distributed workflow, and identify theoretical questions which arise when considering the operation of such a system.

We also present ÜberNet, a Java-based system which implements inter-object communication as a limited form of dynamic distributed workflow. This system, which provides extremely powerful communication capabilities to distributed Java objects, serves both as a proof of concept for dynamic distributed workflow and as a starting point for the future implementation of more complex dynamic distributed workflow systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

As manufacturing and business processes have become increasingly complex, more businesses have deployed workflow management technologies in an effort to reduce costs and streamline operations. Traditionally, such workflow management systems have been restricted to static workflows, and are mainly useful for large companies whose workflows are completely contained within the company itself. With the advent of the virtual organization and virtual corporation, however, a need has developed for workflow management technologies capable of handling dynamic workflows distributed across both physical and organizational boundaries.

In this work, we describe the properties of such workflows and the problems, both implementation and theoretical, which arise when considering them. We focus on the applications of these concepts to the area of distributed objects; when we talk about a workflow occurring within a virtual organization (of people), we are really describing the lower level phenomenon of a workflow occurring among a set of distributed objects (which belong to the people). While these objects and the individuals who own them can interact (an object can, for example, ask its owner to click on an "OK" button to approve a financial transaction), we assume that, primarily, the objects will be acting in an autonomous fashion. This makes the problem of dynamic distributed workflow management and reasoning more a computer science problem than a human-relations problem. However, since human interaction is ultimately required in most workflows, exceptional conditions in a workflow system can be caused by human error or absence, as well as by strictly computational problems.

In this work, we present only an overview of the implementation strategies which can be considered and the theoretical questions which must be answered in order to successfully design and reason about dynamic distributed workflow systems in the distributed objects context. We then examine in detail a small subset of the dynamic distributed workflow problem domain, namely the use of dynamic workflow structures to construct a dynamically reconfigurable communications infrastructure for distributed objects. This is motivated not only by the necessity of starting our exploration with a small piece of the problem at hand, but also by the fact that such a communications infrastructure will be extremely useful during exploration of other parts of the problem domain.

1.2 Workflow and its Applications

The Workflow Management Coalition (WfMC), an international body based in Belgium, has issued a standard glossary of workflow terminology [12], as well as a workflow reference model [11]. Though these are well-defined standards, they apply mainly to workflow in the context of business processes, rather than in the more general sense of workflow among distributed objects which we are exploring here. We will therefore use more simplified and concise definitions of the terms we are borrowing from the general workflow community.

For the purposes of this discussion, we define a *workflow* to be a sequencing of tasks which must be performed in order to accomplish a specific goal. A *task* is an action which must be performed by a single distributed object, and an object which performs such a task is called a *workflow component*. We sometimes refer to workflow components as *workers*.

One of the things which makes workflow a useful concept is that each workflow is reusable; a workflow for processing a loan application at a credit union can be used not only to process more than one loan application, but also to automate the activities of several credit unions. A *job* is the entity which is being processed in a workflow; each loan application, in the aforementioned loan processing workflow, causes a new job to be started.

A workflow *schema* is the actual topology of a workflow, that is, the sequence of tasks which must be performed in order to complete a job. Such schemata can specify the workers which must process a job using well-known identifiers (such as names) associated with specific workers, or using *roles*, attributes indicating the parts played by workers in the workflow. The use of roles allows a workflow specification to be more general. For example, the credit union loan processing workflow might specify either that worker "Bob" is to give final approval to loan applications, or that some worker with the role "loan approval officer" is to give final approval to loan applications. Both specifications work equally well if Bob happens to be in the office, but when he's out sick and Jim is the only loan approval officer in the office, only the second specification will allow the workflow to continue seamlessly.

A workflow schema does not need to be a linear sequence—it can have decision points at which a job takes one of a number of possible paths. More interestingly, it can have points at which a job can *split* into multiple concurrent sub-jobs, and other points at which multiple sub-jobs can *join* to reconstitute a job. As an example of this, consider the following simplified description of a workflow for review of a grant application by the NSF:

1. Application is received by NSF.

2. Review panel of N reviewers is picked from a pool of appropriate reviewers.

3. Application is sent to each reviewer for first stage review.

4. First stage reviews are received and combined into one large document.

5. First stage review document is sent to each reviewer with application, for second stage review.

6. Second stage reviews are received and grant decision is made.

7. Applicant is notified of decision.

In this workflow, there are two split points and two join points; all N reviewers write their first-stage reviews concurrently, the workflow halts until all N first-stage reviews are received, and the same occurs for second-stage reviews. Of course, it is not necessary that split and join points be as symmetric as in this example. In fact, a split point may result in two or more completely different types of sub-job being created. A join point may not only combine two or more completely different types of sub-job, but also wait for some subset of at least M of N sub-jobs to complete, or wait for some other such criteria to be fulfilled. Workflows containing these constructs can become extremely complex, which is why it is necessary to develop methods of reasoning about them.

The applications of traditional workflows—that is, workflows which have static sequencing and are contained within a single organization (such as a corporation)—are well documented. Businesses are currently using traditional workflows for applications from built-to-order computer assembly to loan application processing, and much literature is available on traditional workflows, their applications, and their implementations. The remainder of this discussion is primarily concerned with non-traditional workflows - that is, workflows which are *dynamic*, or *distributed*, or both. A dynamic workflow is one in which the workflow structure can change over time in such a way that the participants in the workflow are changed or the schema of the workflow is modified. In fact, each job in a dynamic workflow can have its own workflow schema, if the system changes rapidly enough. A distributed workflow is one which involves the participation of workers on more than one physical machine, and which is therefore not under the direct control of any one particular machine. In the context of distributed objects and emergent systems, where new objects can appear and old ones can disappear at any time, such dynamic distributed workflows are not only useful, but necessary.

1.3 Overview

The remainder of this thesis is structured as follows. Chapter 2 describes various potential implementation strategies for a dynamic distributed workflow system capable of fulfilling the requirements discussed above, and also briefly introduces some of the theoretical computer science problems which must be addressed when building and using such a system. Chapter 3 details a specific application of dynamic distributed workflow, namely the use of restricted workflows to enable inter-object messaging for emergent distributed systems, which will be the area of concentration for the remainder of the thesis. Chapter 4 describes the design and implementation of ÜberNet, a prototype inter-object messaging package built as a proof of concept for the dynamic distributed workflow approach to inter-object communication, and examines its performance relative to other currently available inter-object communication methods. Finally, Chapter 5 contains some brief concluding remarks.

Chapter 2

Implementation Approaches and Theoretical Questions

In this chapter, we briefly explore various approaches toward the implementation of dynamic distributed workflow systems, and describe some of the theoretical questions which arise when considering their behavior.

2.1 Implementation Approaches

The implementation of a system to support dynamic distributed workflow among objects can be approached in several different ways, but the system must ultimately fulfill certain operational requirements regardless of the choices made during implementation. We now enumerate and discuss these requirements, and propose 3 implementation strategies for a system which can fulfill them.

2.1.1 Requirements Analysis

A dynamic distributed workflow system must meet the following requirements:

Complex workflow schemata. The system must be able to handle workflows where jobs can split into sub-jobs, where sub-jobs can join to reconstitute a job, and where decisions may force the workflow to take one of multiple paths. It must also support schemas which include loops, where the same job is acted upon by the same sequence of workers multiple times until some condition is met.

Dynamic sequencing. The system must allow changes to the sequencing within a running workflow, without causing any jobs to be left in an incomplete or incorrect state. Such changes should be made automatically by the system for purposes of fault tolerance, load balancing, or general efficiency, as well as manually by a workflow designer for the purpose of changing the functionality or other characteristics of a workflow.

Dynamic participation. The system must allow changes to the objects participating in a running workflow, without causing any jobs in process to be left in an incomplete or incorrect state. This includes changes such as direct substitution of one object for another, as well as changes such as replacing a single object with a set of objects (and thereby also changing the workflow schema). If roles, rather than specific object identifiers, are used to determine which objects are participating in a given workflow, then the requirement of dynamic participation implies that objects can change their roles over time.

Distribution of participants. The system must support workflows which contain participants on geographically distributed machines. Thus, it must take into account relevant characteristics of geographically distributed systems (such as potential unreliability of communication channels). If the system is to be used not merely for workflow among distributed objects, but also for workflow including the active participation of human beings, it must also take into account other characteristics of geographical distribution (such as time zones) and be able to cope with the various uncertainties which arise in human interaction (such as a particular worker being out sick one day, or resigning from the company, or being run over by a bus).

In addition, a dynamic distributed workflow system must also meet the following two requirements, which are standard requirements for workflow systems in general:

Concurrency. The system must support multiple jobs running with the same workflow schema simultaneously. This allows a paper review workflow, for example, to control the concurrent reviewing (by possibly distinct sets of reviewers) of multiple papers simultaneously.

Job tracking. The system must be able to provide status information on running jobs, including their current "position" in the workflow. This requires a tracking mechanism sophisticated enough to track the multitude of jobs flowing through the system, but scalable enough to be reasonably efficient.

We now describe 3 implementation strategies for dynamic distributed workflow systems—method call sequencing, self-routing data, and mobile agents—which can potentially meet these criteria. In the course of our description, we will apply each of these strategies to a sample workflow for a one-stage paper review process, which is depicted in Figure 2.1. This workflow is similar to the NSF grant review workflow described in the introduction, but with a single review stage consisting of 2 reviewers.

2.1.2 Implementation Strategy: Method Call Sequencing

One strategy for implementing a dynamic distributed workflow system is to use sequenced method calls to implement the workflows. Assume that we have a number of objects with public interfaces, which are the workers in a workflow system, and furthermore, that we have access to the interfaces we need for the

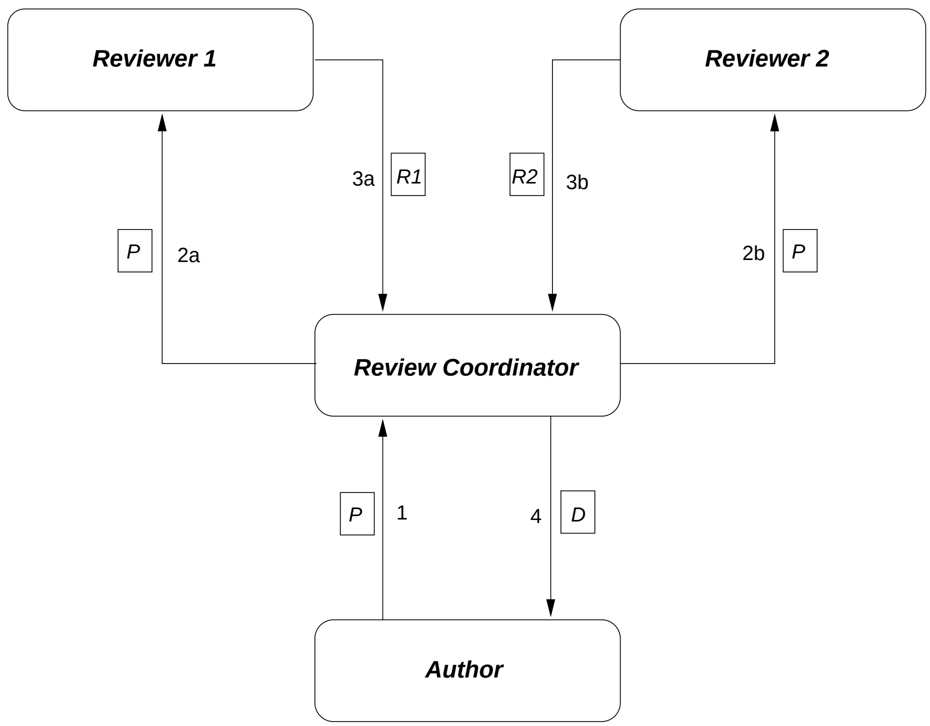

Figure 2.1: Abstract depiction of a simple paper review workflow. Arrows represent communications between workers, which occur in the order indicated by the number next to each arrow. Communications with the same number (i.e.: 2a, 2b) originating from the same source signify a fork in the workflow; communications with the same number arriving at the same destination signify a join. The data elements being transferred as part of each communication, indicated by P, $R1$, $R2$, and D, are, respectively, the paper, the review from reviewer 1, the review from reviewer 2, and the decision made by the review coordinator as to whether or not to accept the paper.

$\{0, \mathit{true} \wedge \neg \mathit{Completed}(0), R1 = \mathit{Reviewer1.reviewPaper}(P)\}$
$\{1, \mathit{true} \wedge \neg \mathit{Completed}(1), R2 = \mathit{Reviewer2.reviewPaper}(P)\}$
$\{2, \mathit{Completed}(0) \wedge \mathit{Completed}(1) \wedge \neg \mathit{Completed}(2), D = \mathit{Coordinator.makeDecision}(P, R1, R2)\}$
$\{3, \mathit{Completed}(2) \wedge \neg \mathit{Completed}(3), \mathit{Author.informOfDecision}(D)\}$

Figure 2.2: Specification of the paper review workflow using the sequenced method call strategy. $\mathit{Completed}(ID)$ is true if execution of the triple with identifier ID has has successfully completed, false otherwise. $\mathit{reviewPaper}$ is a method on a reviewer which takes a paper as a parameter and returns a review, $\mathit{makeDecision}$ is a method on the review coordinator which takes a paper and two reviews as parameters and returns a decision as to whether or not the paper will be accepted, and $\mathit{informOfDecision}$ is a method on the author which takes a decision as a parameter and has no return value.

functions each object must perform. We can then specify and implement the workflow with a sequence of calls to the methods of those interfaces; this allows the individual objects implementing the interfaces to change, so long as the interfaces themselves remain the same.

A central coordinator object, which has an object directory that allows it to find objects which implement specific interfaces, makes all the method calls. There could be one such coordinator object per workflow schema, responsible for tracking multiple jobs, or one coordinator object per job, responsible only for that particular job. In such a system, the sequence of calls for a given workflow could be specified by a list of triples of the form:

$$\{\texttt{ID, Precondition, Method Call}\}$$

where "ID" is a unique identifier assigned to each triple, "Precondition" is a (possibly quite complex) predicate which must hold in order to execute the triple, and "Method Call" is the method call which will be executed as part of the triple.

A basic specification of our paper review workflow using these triples can be found in Figure 2.2. This type of specification allows for forking and joining of jobs, concurrent execution of multiple method calls, looping, and other such constructs; however, the basic specification method we have described here would make some of these constructs cumbersome to implement. Moreover, it would make on-the-fly modifications of the workflow schema difficult, because special steps would need to be devised to modify jobs at each stage of the workflow in order to insure their successful completion. More complex specification methods could simplify such modifications, as well as providing constructs to support fork, joins, looping, etcetera.

The method call system also has the disadvantage of depending on a central coordinator for each workflow schema (or job) to both monitor the progress of jobs and make the actual method calls (depicted in Figure 2.3). This creates a single point of failure, so that even if the worker objects are functioning correctly, a problem with the coordinator object will prevent jobs from progressing through a workflow schema. Still, assuming that we can work around some of these difficulties (by having redundant fallback coordinators, for example), the use of straight method calls is a worthwhile approach to consider for implementing a dynamic distributed workflow system.

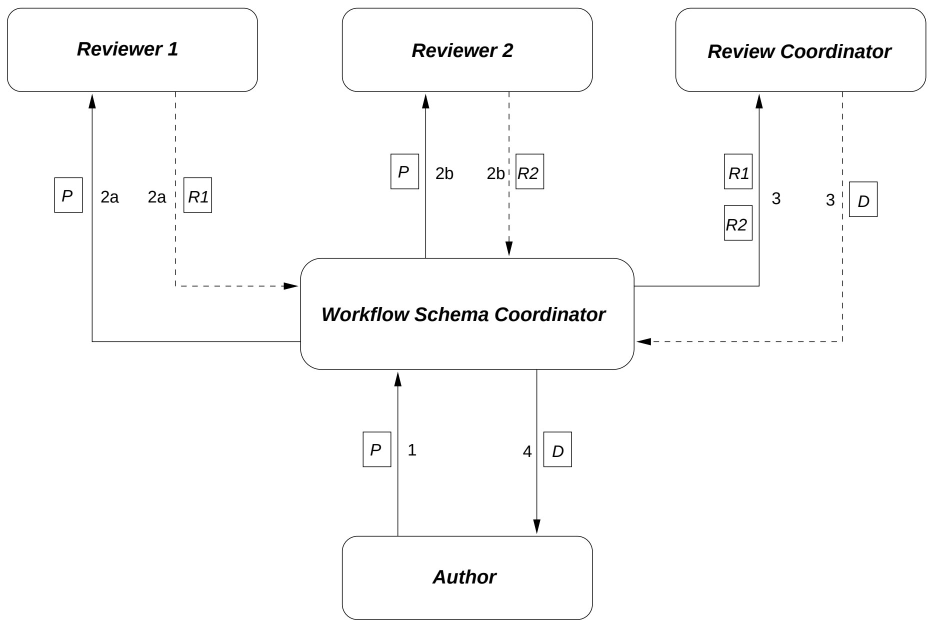

Figure 2.3: Depiction of the paper review workflow implemented using the sequenced method call strategy. Solid arrows represent method calls, which occur in the order indicated by the number next to each arrow, while dashed arrows represent return values from these calls. Method calls 2a and 2b occur concurrently. The data elements being transferred as part of each method call, indicated by P, $R1$, $R2$, and D, are, respectively, the paper, the review from reviewer 1, the review from reviewer 2, and the decision made by the review coordinator as to whether or not to accept the paper. Note that we have chosen to start the workflow with a method call made by the paper author on the workflow schema coordinator, and to end it with a method call made by the coordinator on the author.

2.1.3 Implementation Strategy: Self-Routing Data

Another strategy for implementing a dynamic distributed workflow system is to use a message-passing system with self-routing data. Each job could consist of a message which contains not only the data necessary for the job, but also all the other information necessary to complete the workflow schema. If we assume that, in such a system, each object has the capability to "look up" other objects by the semantic content they are able to handle (not a small assumption), and that each object performs exactly one type of processing on each semantic data type it accepts, we can use these self-routing messages to implement a dynamic distributed workflow.

A basic implementation is fairly straightforward: The workflow is started by some central workflow coordinator, which creates a message containing not only all the initial conditions of the workflow, but also a compact specification of the workflow schema and some state information, including an identifier which ties the message to a specific workflow instance. The specification could be in the form of a list of pairs, with each pair having the form:

$$\{\texttt{ID, }\{\texttt{Condition-Destination Triples}\}\}$$

where "ID" is a unique identifier assigned to each pair, and "Condition-Destination Triples" is a list of triples of the form:

$$\{\texttt{Condition, Destination, ID}\}$$

where "Condition" is a predicate on the data being processed, the state of the job and the state of the worker objects, "Destination" is the next destination for the workflow message (specified by semantic capability and possibly also by other information), and "ID" is the identifier corresponding to the pair (in the workflow specification) which will be "executed" at the destination.

The state information contained in the message includes any semantic data which is generated and/or manipulated during the workflow, as well as the identification number corresponding to the pair from the workflow specification which is to be executed when the message arrives at its destination. A basic specification of our paper review workflow using these constructs appears in Figure 2.4, and the message passing which occurs during the workflow is depicted in Figure 2.5.

When a message arrives at a given destination object, its contents are examined and appropriate processing is performed according to their semantic type (recall that each object performs exactly one type of processing on each semantic data type). Then, the object looks to see which pair from the workflow specification list must be executed, and processes all the condition-destination triples within that pair in turn. For each triple whose predicate holds, the object sends the message (with updated state) to the corresponding destination, setting the "to-execute" ID in the message to that indicated within the triple. Once this processing is complete, it sends a notification to the workflow coordinator, informing it that the workflow message has been processed.

$$\begin{array}{l}
\{0, \{\{true, \mathit{Reviewer1}, 1\}, \{true, \mathit{Reviewer2}, 2\}\}\} \\
\{1, \{\{\neg(R1 = null), \mathit{ReviewCoordinator}, 3\}\}\} \\
\{2, \{\{\neg(R2 = null), \mathit{ReviewCoordinator}, 3\}\}\} \\
\{3, \{\{\neg(D = null), \mathit{SchemaCoordinator}, 4\}\}\} \\
\{4, \{\{true, \mathit{END}, null\}\}\}
\end{array}$$

Figure 2.4: Specification of the paper review workflow using the self-routing data strategy. The *Reviewer1* and *Reviewer2* objects generate reviews when they receive the self-routing data; the *ReviewCoordinator* object generates a decision once it has received both reviews. *END* is a special destination which tells the workflow schema coordinator that the workflow is complete and that it should return the decision to the author; interactions between the author and the workflow schema coordinator are not specified here.

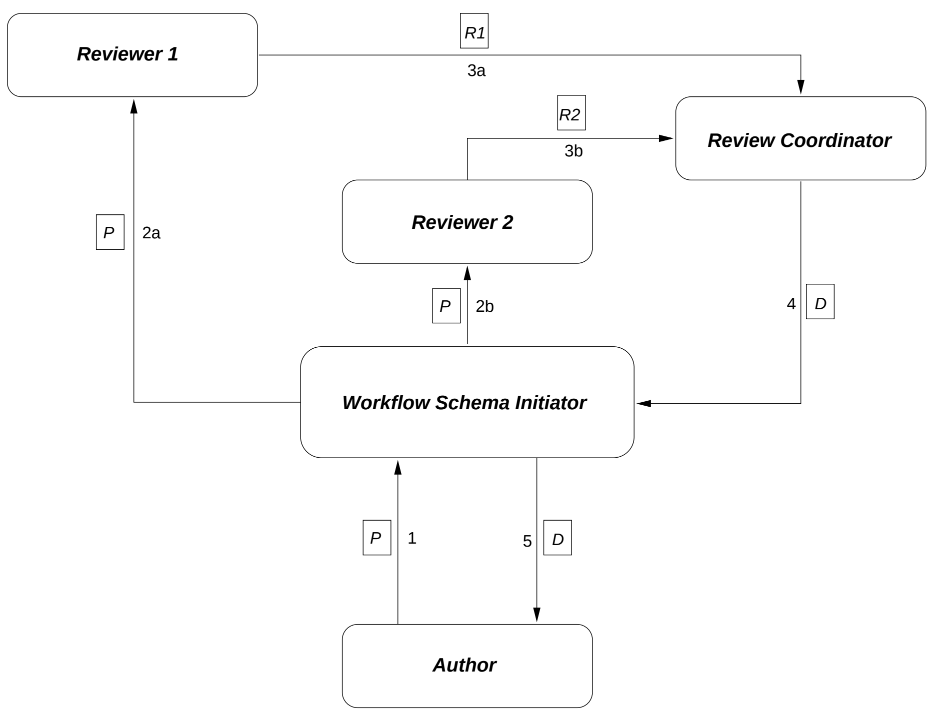

Figure 2.5: Depiction of the paper review workflow using the self-routing data strategy. Arrows represent messages sent between workers, which occur in the order indicated by the number next to each arrow. Messages with the same number (i.e.: 2a, 2b) originating from the same source signify a fork in the workflow; messages with the same number arriving at the same destination signify a join. The data elements being transferred as part of each method call, indicated by *P*, *R1*, *R2*, and *D*, are, respectively, the paper, the review from reviewer 1, the review from reviewer 2, and the decision made by the review coordinator as to whether or not to accept the paper.

Such a specification allows for forking and joining of jobs; forking is implemented easily, by adding multiple condition-destination triples which have the same predicate, while joining is implemented with a predicate on the number of component sub-jobs received. Other constructs, such as looping, are also implemented easily. It also allows for modification of workflow schemas on the fly, but in a different sense than the method call system—any workflows already in process when the workflow schema is changed will continue using the old schema, as they are now self-routing and not controlled by a central coordinator. While this can be seen as a possible disadvantage, given our requirements, it is certainly a viable solution in many cases, and given sufficiently savvy worker objects, on-the-fly schema modification would be relatively straightforward to implement.

The self-routing data strategy has the advantage of scalability—the central coordinator is now used only as a storage point for schemas and an initialization point for workflow instances, and once a workflow instance has been started, it wends its own way through the workflow system until it is completed (or fails catastrophically). Given reliable communication mechanisms, and the previously mentioned semantic capability directory of objects, it has great potential as a basis for implementing a dynamic distributed workflow system.

2.1.4 Implementation Strategy: Mobile Agents

A third strategy for implementing a dynamic distributed workflow system involves the use of *mobile agents*. A mobile agent, for the purposes of this discussion, is a piece of software which actively migrates among multiple machines and uses a standardized API to interact with software components running on those machines. The use of mobile agents in static distributed workflow systems has been previously discussed, in [7].

Assume that we have a set of workers with public interfaces which contain both semantic and syntactic information (as in the method call sequencing strategy). Moreover, assume that we have a *mobile agent platform*, a software framework which facilitates the migration of agents to the machines containing the worker objects and provides access to the worker objects via a standardized API. Such an agent platform might give agents the ability to obtain local references to the objects on each machine to which they migrate and thereby interact with the objects directly, or it might restrict agents to interact with objects indirectly, through some security mechanism built into the platform. These two modes of interaction are fundamentally different, both in terms of security and in terms of semantics, and the decision as to which one would be better for the purposes of a dynamic distributed workflow implementation will be left as a future research question. Regardless, with such a workflow platform available, we can implement a dynamic distributed workflow system in which each job is represented by a mobile agent, as follows:

An agent is created with a complete picture of the workflow schema, and the initial data required to begin the job. The agent then migrates to whatever machine it has to in order to execute the first step of the workflow schema. When this step is complete, it then migrates to wherever it must in order to complete the next step, repeating this process until the job has been completed. Forking and joining within a workflow schema are handled by the creation of sub-agents at a fork point, and the collection of these sub-agents (i.e.: waiting for all of them to arrive) at the corresponding join point. Because the agent controlling the job encapsulates both the data and the workflow schema, there is no need for any central coordination point beyond the fact that the agent must be created somewhere, and must report the completion status of the job somewhere. A mobile agent implementation of our paper review workflow which illustrates this appears in Figure 2.6.

We assume that any mobile agent framework which would be used for this purpose would include facilities for communicating with an agent without needing to know its current location. Given this capability, we can track the status of individual jobs simply by communicating with their associated agents at whatever time intervals we desire. This also allows us to make changes to the workflow schema and have them affect even running jobs, by forwarding those changes to the agents associated with those jobs and allowing the agents to "decide" on an appropriate course of action.

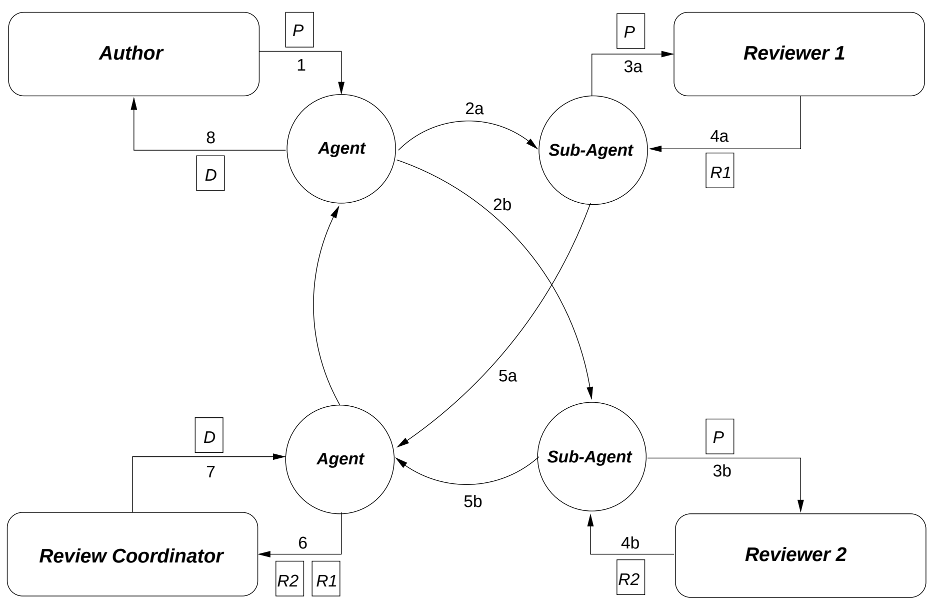

Figure 2.6: Depiction of the paper review workflow using the mobile agents strategy. Straight lines represent communication between agents and worker objects, while arcs represent agent migrations. Communications and migrations occur in the order indicated by the number next to each arrow. The letter (i.e.: a, b) after a numbered step indicates which sub-agent is handling that step; steps 2a and 2b signify the forking of the initial agent into two sub-agents, while steps 5a and 5b signify the rejoining of the two sub-agents. The data elements being transferred as part of each communication, indicated by P, $R1$, $R2$, and D, are, respectively, the paper, the review from reviewer 1, the review from reviewer 2, and the decision made by the review coordinator as to whether or not to accept the paper.

There are difficulties associated with this implementation strategy, however. One of these involves security—the organizations whose objects will be participating in the workflow system may be reluctant to allow mobile code from outside sources to execute on their systems. Another difficulty is the design of the mobile agent system itself, which we have left completely unspecified except for some general characteristics. However, assuming these difficulties can be overcome, the mobile agent approach may be the most appropriate for dynamic distributed workflow systems.

2.2 Theoretical Questions

The success of the implementation strategies discussed above hinges on the answering of many theoretical questions which arise when considering the behavior and specification of dynamic distributed workflow systems, the objects which participate in them, and the jobs which run in them. This section enumerates and briefly discusses some of the more interesting of these questions.

2.2.1 Specification

The question of specification arises at multiple levels when considering a dynamic distributed workflow system. We have already briefly discussed first-pass specification methods for workflow schemata in two of the three implementation strategies in the previous section, but it is clear that these specification methods are far from optimal. A better specification method would be intuitive enough to allow the use of complex constructs such as forks, joins and loops without having to resort to "goto"-style flow control. Such a specification method should be at a higher level than that which is actually used by components in the workflow system, to allow proofs about the behavior of a specific workflow schema to be more easily carried out; as long as there is a well defined (and rigorously proven) way to translate a high-level workflow schema specification to low-level object control code, it makes much more sense to use a high-level specification as a basis for reasoning about a given workflow schema.

Some work has already been done in the area of specifying workflows which involve concurrent actions, as in [1], and in the area of workflow evolution (the component of dynamism which involves the changing of schemata while they are operational), as in [2]. Developing a specification method for workflow schemata in dynamic distributed workflow systems must build upon this work and upon the work already done in specification methods for traditional workflow schemata.

In addition to specification of the workflow schemata, a specification method for individual jobs is also necessary in order to reason about the activity occurring with relation to a particular job in a workflow system. Development of a specification method at the level of worker roles is also necessary, as it would be impossible to create a successful role-based workflow system without such a specification method.

2.2.2 Composition

The question of composition is a natural one to consider when working with interorganizational workflows. It is quite possible that an organization which needs to perform a task in such a workflow may not want to reveal to the workflow designer exactly how it goes about accomplishing this task, because it may involve proprietary information. Thus, the final workflow may end up as a composition of multiple workflows, about which only certain characteristics which have nothing to do with their internal structures, such as input and output requirements, are known. There are many other reasons why composition of workflows might be useful, such as for building up extremely large scale workflows out of well-established, verified, smaller workflows—since small workflows will clearly be easier to specify and prove than larger ones, this approach may save considerable time and labor.

2.2.3 Atomicity

The question of atomicity becomes an important one when considering workers which may be participating in multiple concurrent workflows. It must be possible in certain circumstances to guarantee atomicity of sub-jobs, if not of entire jobs, to ensure correct operation of workflows which overlap in either membership or in the data on which they act. Atomicity, in this context, has the following implication: once an atomic sub-job starts, all other jobs and sub-jobs must wait until after the atomic sub-job has been completed to execute any operation which could potentially alter the atomic sub-job's outcome. While traditional transactional methods may be sufficient for this purpose, it is in no way obvious that they are, and entirely new methods of ensuring atomicity may be required to handle the atomicity requirements of dynamic distributed workflow systems.

2.2.4 Large-Scale Reasoning

Reasoning about the behavior of even one job running with a particular workflow schema may be difficult. However, reasoning about the large-scale behavior of a dynamic distributed workflow system containing tens or hundreds of different workflow schemata with hundreds or possibly thousands of jobs running simultaneously will likely be orders of magnitude more difficult. Thus, the investigation of large-scale reasoning methods, which allow straightforward reasoning about the behavior of such systems, is necessary to insure that we will be able to understand what a dynamic distributed workflow system will do once it is started and multiple concurrent jobs begin running within it.

Chapter 3

Inter-Object Messaging as Dynamic Distributed Workflow

In the previous 2 chapters, we have presented an overview of the concept of dynamic distributed workflow and discussed implementation approaches and theoretical questions related to a full-fledged dynamic distributed workflow system. We now discuss a much narrower problem, that of inter-object messaging, and explore how the concept of a dynamic distributed workflow can help in devising a good solution to this problem. This is important not only because inter-object messaging is an interesting area in its own right, but also because a good inter-object messaging system is necessary to construct dynamic distributed workflow systems such as those previously described.

3.1 The Problem: Inter-Object Messaging

The problem of inter-object messaging arises in every distributed object infrastructure in which the objects must communicate with each other. Some, such as the Object Management Group's CORBA [8] and Sun's Java Beans [9], use method call semantics to hide the underlying messaging protocols which are used to allow inter-object communication. Others, such as version 1.0 of the Infospheres Infrastructure [4], use direct messaging with the concept of mailboxes and object addresses.

The basic idea is that an object must be able to send data (and, in some more complex systems, code) to other objects in the distributed system in a format which allows it to be understood by the receiving objects. Assuming that such distributed systems are composed of objects running on multiple hardware platforms, this involves such low-level concerns as byte-swapping (for conversion between little-endian and big-endian systems), and other, higher-level concerns such as message typing and the communication of useful semantic data. Message security may also be a concern in corporate or personal systems which deal with sensitive information, and the reliability of the messaging system must be well-specified so that the semantics of inter-object communication are well understood.

In all the previously-mentioned systems, these problems are solved by using predefined, proprietary protocols for communication between objects. These protocols are static, and cannot be changed without upgrades to the systems in question. Moreover, when they are changed, it is usually at the expense of backward compatibility, and this can cause serious problems unless all the systems hosting these distributed objects and their support structure are upgraded simultaneously. In a distributed system of the future, which may consist of millions of objects running on thousands of machines scattered around the globe, it is clearly infeasible to use a system which requires such simultaneous updates to its support software.

However, it is possible to use the concept of dynamic distributed workflow as a basis for developing an inter-object messaging system which does not exhibit these problems, where enhancements to communication and security protocols need not come at the expense of cutting currently-running objects out of the communications loop. A communication system designed using a dynamic distributed workflow approach could be capable of adapting on-the-fly to changes in communication protocols and other infrastructure alterations.

3.2 Applying Workflow Concepts to Messaging

In order to design such a communication system, we must first determine exactly how to view the inter-object communication problem in terms of dynamic distributed workflow. Communication of a message from object A to object B requires multiple steps to be performed by object A and its supporting infrastructure, namely:

1. Gather the data to be communicated.
2. Encode the data into a form which can be sent easily over the network.
3. Send the data to the physical machine on which object B is running.

When the data actually gets to the machine on which object B is running, multiple steps must be performed by object B's supporting infrastructure in order for the message to be delivered:

1. Determine which object the message is meant for.
2. Decode the data so that it can be understood by object B.
3. Give the data to object B in whatever format it expects.

All these steps can be viewed as a single workflow, which delivers a particular message from object A to object B. Breaking the steps into further sub-steps, we could isolate the parts of the message encoding and sending which are common to all messages (not just those from object A to object B), and create generic components for use in dynamic distributed workflows. Such workflows could then be created on-the-fly whenever any object needs to send messages to any other. Moreover, we can make the coordination objects of the workflow system sophisticated enough to automatically detect the use of new communication or security protocols, and to ensure that the correct workflow components are available whenever and wherever they are needed. An illustration of message flow through such a system appears in Figure 4.1, during our discussion of the messaging model for our implementation of these concepts.

This type of dynamic distributed workflow is substantially more basic than what has been discussed in previous chapters. It involves primarily linear workflow schemata, with no forks or joins, which considerably simplifies reasoning and implementation. It also involves "small" worker components, which are responsible not for large-scale business processes, but rather for simple message processing tasks. Nevertheless, to make such a system work involves many complex design and implementation details, as we shall see in Chapter 4.

3.3 Benefits of a Workflow-Based Messaging System

The implementation of inter-object communications with a system based on this dynamic distributed workflow concept brings with it multiple benefits. Foremost among these is the ability to add new protocols, such as novel message encryption and compression methods, to the system and have all running objects instantly gain the capability to "understand" messages sent using these new protocols.

One consequence of this is that with such a system, to update the version of a particular protocol being used, only the sender of a message needs to have the updated version—the receiver will always be able to understand everyone's messages, regardless of the protocols actually pre-installed on the receiver's machine.

Another consequence is that organizations are free to implement and use proprietary protocols. They can then choose to make the relevant worker components available for use by other organizations, and therefore gain the ability to communicate inter-organizationally using their proprietary protocols. They can also employ firewalls or key-based security mechanisms to ensure that their protocols are used only in an authorized fashion, so that the protocols do not become widely accessible and lose their value to the organization.

An important benefit of a workflow-based communication system arises from the fact that one can use communication workflows to specify and change the quality of service an object receives in its communications, while the object is running. For example, communications between objects A and B might need to be triple-encrypted for security purposes, while communications between objects A and C might need only to be as fast as possible. Object A would then have two separate communication workflows, one for communications with object B and one for communication with object C, which would provide these two different levels of service. This variable quality of service allows for custom structuring and tuning of communications in an entire distributed system to optimize performance.

As we can see, such a system would be extremely useful in generating structured interaction among distinct organizational object systems. In the next chapter, we discuss the design, implementation, and performance of a prototype system based on these ideas.

Chapter 4

ÜberNet: The Infospheres Network Layer

In this chapter, we discuss the design and implementation of ÜberNet, an inter-object communication framework designed using dynamic distributed workflow concepts. We also evaluate the performance and feature set of ÜberNet as compared to other available inter-object messaging solutions.

4.1 System Requirements

In designing ÜberNet, there were three basic requirements which had to be satisfied, as follows:

Platform independence. In order to be a successful proof of concept for the idea of a messaging system based on the idea of dynamic distributed workflow, ÜberNet had to be able to run on multiple hardware platforms without needing to be customized for each one.

Dynamic extensibility. Emergent distributed systems can change over time, in ways which may not be anticipated when the systems are created. A messaging system for objects in such a system must be able to adapt to changes in the system by extending its capabilities to include new messaging protocols without requiring explicit updating or reinstallation.

Simplicity. Because distributed systems are so complex in their own right, the communication framework supporting these systems should be as simple as possible, so that programmers can understand it more easily. Additionally, a simple underlying message layer offers fewer opportunities for the distributed system as a whole to behave in an unpredictable fashion.

In addition to these basic requirements, the following "higher-level" design goals also played a large part in determining the design of ÜberNet:

Compositionality. Since ÜberNet was to be a proof of concept for dynamic distributed workflow, it needed to incorporate compositional notions at least insofar as all dynamic distributed workflow systems must be compositional. One design goal for ÜberNet was to incorporate compositional notions not just to the extent necessary, but to the greatest extent reasonably possible. This goal was set in the belief that a highly compositional system would be easier to build on in the future, as well as easier to use in the present.

Compatibility with the Infospheres Infrastructure. Since development on ÜberNet and on version 2.0 of the Infospheres Infrastructure (II) was occurring concurrently, it was decided that ÜberNet should be able to serve as the underlying communication layer for the II. This would allow writers of Infospheres objects to have all the advantages of ÜberNet's dynamically reconfigurable messaging system if they chose to, as well as potentially improving communications efficiency for the Infospheres Infrastructure's internal communications. More information on the synergy between ÜberNet and the II is available in [5].

4.2 Implementation Language

The choice of implementation language was perhaps the most important design decision, as it determined what language facilities would be available for use by the ÜberNet system. The system requirements which played the largest role in determining the implementation language were platform independence and dynamic extensibility. Version 1.1 of the Java programming language was chosen for the implementation of ÜberNet, because its platform-neutral nature makes it ideal for cross-platform distributed object computing, and because it provides several mechanisms which make the implementation of a dynamically extensible system far more straightforward than in other development languages. Additionally, the choice of Java made integration with version 2.0 of the Infospheres Infrastructure (which is also written in Java) far easier than it would have otherwise been. We now briefly describe the features of the Java language which were most conducive to the implementation of ÜberNet.

Platform independence. Java programs are compiled to a platform-neutral bytecode, which is executed at runtime by a Java virtual machine. Since Java virtual machines exist for all of the commonly-used hardware platforms, an inter-object communication system written in Java allows communication among objects on all of those platforms. If the objects are written in Java, such communication is straightforward; if not, then Java "wrapper" objects can be written to provide a bridge between the non-Java objects and the communication system.

Runtime class loading. Java's class loading mechanism allows a developer to specify, at runtime, the actual classes which are to be used for certain operations. Using a special subclass of Java's `ClassLoader` class, the bytecode for an arbitrary component can be loaded at runtime from an arbitrary location. This location is specified by a URL, which means that the bytecode can be stored on a web server, an FTP server, the local filesystem, or some other type of server for which a well-established URL scheme exists. This allows the dynamic extensibility required in the ÜberNet system, by allowing new protocol modules to be downloaded from arbitrary sources and used at runtime when they are needed. The security concerns which inevitably arise when dealing with code from a foreign machine can be addressed by using Java's security model, which allows for digitally signed bytecode and prevents remotely-loaded code from doing damage to the local system by means of a sandbox model. More detailed information on Java's security architecture can be found in [6] and [3].

Object serialization. Java 1.1's object serialization mechanism allows any object implementing a special interface (called java.io.Serializable) to be converted to and from a standard data representation format, capable of being stored on disk or transmitted over a network. This provides a straightforward way of sending messages—a message can be an arbitrary object implementing the java.io.Serializable interface, and they can be communicated using Java's standard object serialization mechanisms (with a few slight modifications, which will be discussed later).

Object reflection. Java 1.1's reflection mechanisms allow an object to discover information at runtime about the interfaces of objects which were totally unknown at compile time. This allows the ÜberNet system to interact with new protocol stack modules properly when it encounters them, and also allows objects communicating via ÜberNet to receive messages of arbitrary types and still be able to "make sense" out of them.

Networking support. Java includes class libraries for working with network communications via TCP, UDP, and IP Multicast in a straightforward and clean fashion. The presence of these networking libraries on the Java platform, especially the integrated support for IP Multicast, eliminates the need to work with these protocols directly at a lower level; this removes a great deal of the complexity inherent in the implementation of any system which uses networked communications.

4.3 Design Choices

There were several major design choices which shaped the overall design of ÜberNet, ranging from the approach taken toward workflow structures to the mechanism for message processing within the system. This section describes these design choices.

4.3.1 Workflow Structures

In Chapter 3, we discussed how a communication among objects could be viewed as a dynamic distributed workflow. Basically, the "steps" of the communication would be split into parts, and these parts would be directly translated into workflow components. The actual workflow schemata within ÜberNet are hybrids—they contain elements of both the self-routing data approach (discussed in Chapter 2) and the static approach of traditional workflows. Each message in ÜberNet is effectively a job with its own, possibly unique, workflow schema.

The ÜberNet design includes three main component types: *sending protocol modules*, *receiving protocol modules*, and *wire protocol daemons*. Sending protocol modules process outgoing message data before it is sent over the network, while receiving protocol modules process incoming message data after it is received from the network. For every sending protocol module, there is a corresponding receiving protocol module. Wire protocol daemons are responsible for actually transmitting data to and receiving data from the network using low-level protocols (such as UDP).

The protocol modules on the sending side of a particular channel are arranged into a *protocol stack*. This stack consists of whichever protocol modules are necessary for the particular message processing being done—examples include an encryption module, a message reliability module, and a message segmentation module. At the bottom of each protocol stack is a wire protocol daemon. Each protocol stack on the sending side is the static part of the channel's workflow schema—it does not change, except when the communication semantics of the channel are changed by the object controlling the channel.

The receiving side of a communication channel has no predefined protocol stack. Instead, for each message that arrives, the correct sequence of components which must process the message is determined on-the-fly, using annotations generated by the sending side components at the time the message is sent. The flow of a message through the system is shown in Figure 4.1.

If a message arrives for which ÜberNet cannot find the required receiving module locally, it uses information in the message headers to obtain the required module from a remote machine via the World Wide Web or FTP. This ensures that the complete workflow schema for a channel can always function correctly, and that messages can be correctly received regardless of the pre-installed capabilities of the receiving object's runtime system.

4.3.2 Messaging Model

ÜberNet's messaging model is based on the concept of messages and mailboxes. A subset of this messaging model was previously used in designing the communication layer for version 1.0 of the Infospheres Infrastructure.

In this model, each object has a self-defined set of named mailboxes, and can create or destroy mailboxes at any time. Mailboxes are protocol stack modules which are special, in that they are always at the top

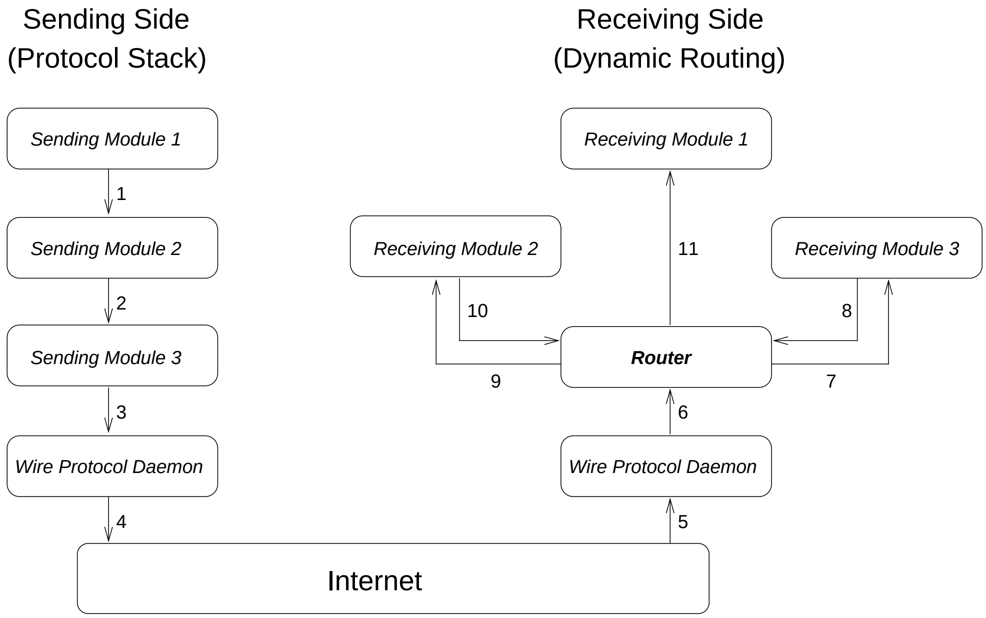

Figure 4.1: Example of message flow in ÜberNet. The message travels along the path shown by the numbered arrows. Sending and receiving modules with the same number correspond to each other.

of the stack. Each mailbox can either send or receive messages, but not both; a mailbox which sends messages is called an *outbox*, and a mailbox which receives messages is called an *inbox*. There are separate namespaces for inbox and outbox names, but each namespace is at the level of the network interface—it is not possible for two objects using the same network interface to have inboxes (or outboxes) with identical names. A communication channel is formed by registering an inbox (by name) with an outbox—this is called "binding" the outbox to the inbox.

In ÜberNet, an outbox can be bound to one or more inboxes (as in Figure 4.2), as well as to one or more IP Multicast groups. An arbitrary number of outboxes can be bound to the same inbox. There is no way for an inbox to specify which outboxes are bound to it, though it can filter incoming messages to eliminate those from specific senders or of undesirable types. An inbox must, however, explicitly join any IP Multicast groups in which it wishes to participate. Therefore, the construction of a point-to-point communication channel is the sole responsibility of the object on the sending side of the channel, while membership in an IP Multicast group is the responsibility of both sending and receiving objects in the group. IP Multicast group communication is illustrated in Figure 4.3.

On the sending side, a message is sent by depositing it into an outbox. A copy of the message is sent to each inbox to which the outbox is bound, and to each IP Multicast group of which the outbox is a member, at the time the message is deposited. An outbox is a first-in first-out queue; two messages deposited in an outbox are always sent in the order in which they are deposited. On the receiving side, a message is received by removing it from an inbox. An inbox is a first-in first-out queue; two messages arriving at an

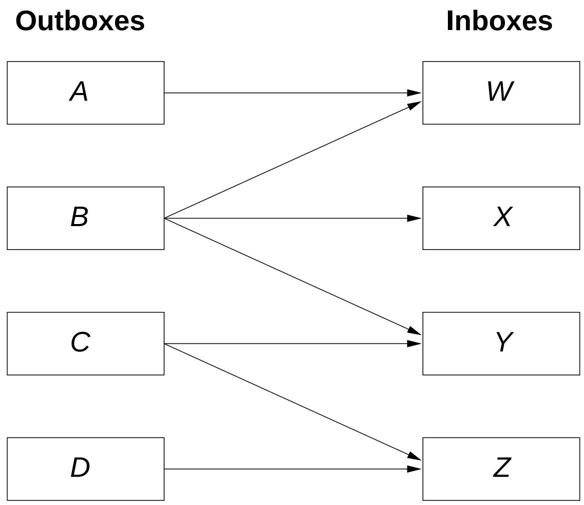

Figure 4.2: Binding of outboxes to inboxes in ÜberNet. Arrows indicate bindings; outboxes B and C are bound to multiple inboxes.

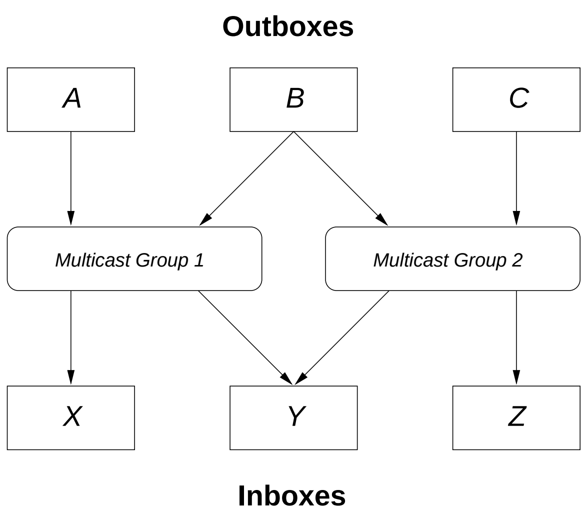

Figure 4.3: IP Multicast group communication in ÜberNet. Outbox B is broadcasting to two multicast groups, and inbox Y is listening to two multicast groups.

inbox are always removed from the inbox in the order in which they arrive.

Because ÜberNet supports modular protocol stacks, the fact that outboxes and inboxes are first-in first-out queues does not mean that message ordering is guaranteed; the semantics of each individual channel are determined by the protocol stack being used on the sending side of that channel. This makes the messaging model quite flexible, as it allows for both point-to-point and group channels with various semantics, while still retaining the ease of design and use provided by familiar concepts such as mailboxes and messages.

4.3.3 The Java Beans Model

As mentioned previously, the protocol stacks in ÜberNet are composed of protocol stack modules and wire protocol daemons. In order to ensure that these components would be interoperable, it was decided to implement them according to the Java Beans component model as specified by Sun in [9]. This model specifies naming conventions for methods and attributes of components, in such a way that Java Bean manipulation tools can discover enough information about a particular Bean by introspection to be able to work with it in reasonable ways.

The use of the Java Beans model means that the components communicate with each other using *events*, which are special Java objects used as the arguments to specially-named method calls in the components' interface. Components on the sending side use *message sending events* to communicate message data, while components on the receiving side communicate message data using *message receiving events*. These events encapsulate the actual message data being communicated among the components, as well as other information about the message (such as its timestamp, its message ID, the address of the mailbox from which it originated, etcetera).

One important benefit of using the Java Beans component model is that visual application building tools, such as Sun's Java Studio [10], can be used to visually compose Beans to create complete applications. Thus, using ÜberNet with other Java Beans, it is possible to visually build not only an application, but also the communication channels to be used by that application. This allows the distributed system designer to have a clearer idea of exactly what sort of processing is being done on messages traveling through the system, and to change this behavior on-the-fly simply by moving some icons around in an application building window.

4.3.4 Message Processing and Java Streams

In order to allow the ÜberNet protocol stack modules to process messages, a standard means of manipulating message data was necessary. We decided to use Java's standard stream mechanism as a way of allowing the protocol stack modules to perform arbitrary processing on messages.

Java's stream mechanism is compositional, in the sense that streams are composed together to combine their functionality. For example, if we start with a ByteArrayOutputStream, we have a stream which takes all the input it receives and writes it to a byte array. We can then place a DeflaterOutputStream atop our already-existing ByteArrayOutputStream, to get a stream which first compresses (deflates) the input it receives, and then writes it to a byte array. Finally, we can place a ObjectOutputStream atop our composed stream, to get a stream which takes objects as input, serializes them, compresses them, and then writes them to a byte array. We have now composed 3 streams to make one stream which performs multiple functions.

In ÜberNet message sending and receiving events encapsulate such "stacks" of Java streams. At message initialization time, each module in the sending side protocol stack can add an output stream to the stack of streams in the message sending event, allowing any sort of processing of which a Java stream is capable to be performed automatically when data is written to the message after initialization. Then, when the message arrives at its destination, the receiving side protocol stack modules add the corresponding input streams to the stack of streams in the message receiving event, to properly decode the message.

If a particular protocol stack module needs to perform more complicated processing than that which can be encapsulated in a Java stream, it can do that as well by "masquerading" as the bottom of the protocol stack, getting the complete message, and then sending the message on to the actual bottom of the protocol stack. The protocol stack modules which perform reliable ordering and segmenting (described in the next section) exhibit this type of behavior.

4.4 Implementation

In this section, we describe the implementation of ÜberNet, including a summary of the included components and extension mechanisms. This section does not contain any information on programming with ÜberNet–such information can be found in the ÜberNet User's Guide [13]. Complete API documentation for ÜberNet is available online from the Caltech Infospheres Group web server, *http://www.infospheres.caltech.edu/*.

4.4.1 Wire Protocol Daemons

ÜberNet includes four wire protocol daemons, which enable communication via TCP, UDP, IP Multicast, and Loopback. Each of these provides different communication abilities, as follows:

4.4.1.1 TCP

The TCPDaemon is the wire protocol daemon which enables communication via the TCP/IP protocol. It handles both incoming and outgoing messages.

TCP has the advantage that it provides reliability and ordering of messages for free. However, this advantage comes at the price of overhead incurred by setting up (and tearing down) connections and by the TCP/IP protocol itself. Since TCP is connection-based, the TCPDaemon must open and maintain connections to remote TCPDaemons in other ÜberNet systems. There is a hard limit to the number of open TCP connections in any given operating system; the TCPDaemon therefore uses a small maximum number of connections, and closes the least recently used connection whenever a new connection is needed and the maximum number has already been reached. This enables connections between frequently-communicating objects to persist for long periods of time, reducing connection setup overhead.

4.4.1.2 UDP

The UDPDaemon is the wire protocol daemon which enables communication via UDP/IP datagrams. Like the TCPDaemon, it handles both incoming and outgoing messages.

UDP is a connectionless protocol, so it offers the advantage of speed. Since it is not necessary to establish a connection with each destination host or to keep track of open connections, much of the overhead of TCP is not incurred by UDP. However, UDP datagrams are neither reliable nor ordered, so extra steps must be taken to ensure that a message sent via UDP actually reaches its destination. Protocol stack modules which do exactly this are part of the standard ÜberNet system, and are described later in this chapter. UDP datagrams sent by the UDPDaemon are limited to a maximum size of approximately 16 kilobytes, to improve both communication and memory allocation efficiency; a protocol stack module which segments messages into multiple small pieces is also part of the standard ÜberNet package.

In addition to handling all incoming and outgoing UDP message traffic, the UDPDaemon also handles all outgoing IP Multicast traffic, since IP Multicast packets are simply UDP datagrams sent to a multicast address.

4.4.1.3 IP Multicast

The MulticastDaemon is the wire protocol daemon which enables communication via IP Multicast. It handles only incoming messages directly, but is also responsible for managing membership in multicast groups. The MulticastDaemon is tightly integrated with the UDPDaemon.

IP Multicast allows single datagrams to be sent simultaneously to many listeners with a single send operation, rather than a single send operation per datagram destination. This can dramatically improve the efficiency of group communications, by reducing both the processing time necessary on the sending side and the network bandwidth required per message. These efficiency improvements come at a price, however: like UDP, IP Multicast provides no guarantees about message reliability or ordering, and it is far more difficult to provide such guarantees on top of IP Multicast than on top of UDP. There are no reliable multicast facilities in the standard ÜberNet implementation.

ÜberNet provides support for multicast group membership, as detailed in the descriptions of the top-level messaging components in Section 4.4.2.1. Additionally, methods are provided to allow individual protocol stack elements to join and leave multicast groups. The MulticastDaemon tracks the number of requests made to join or leave each multicast group, and opens and closes multicast sockets as necessary.

4.4.1.4 Loopback

The LoopbackDaemon, while technically a wire protocol daemon, does not actually implement an on-the-wire protocol. It takes outgoing message events and converts them to incoming message events within the same virtual machine. All destination addresses are treated identically by the LoopbackDaemon—a message addressed to mailbox "Valen" on some other virtual machine is delivered to mailbox "Valen" on the LoopbackDaemon's virtual machine (and, moreover, its message ID is modified to make it appear to have originally been addressed to the LoopbackDaemon's virtual machine).

The LoopbackDaemon allows for the efficient testing of "distributed" system configurations on a single virtual machine, without the use of a network interface. This is especially useful if no working network interface is available, as on a disconnected laptop system.

4.4.2 Protocol Stack Modules

ÜberNet includes five pairs of protocol stack modules, one of which (SimpleSender and SimpleReceiver) is a bare-bones example which does no actual message processing.

4.4.2.1 Inbox & Outbox

Inbox and Outbox are the top level protocol stack modules which implement the mailboxes of the ÜberNet messaging model. Together, they allow the communication of any object which implements the java.io.Serializable interface (that is, any Java object which supports Java's standard object serialization interface) among multiple virtual machines.

An Outbox, when constructed, is assigned a unique (within its virtual machine) name, either from a parameter passed at construction, or automatically from a set of unique mailbox names reserved by ÜberNet for internal use. Once created, an Outbox can be bound to an arbitrary number of InboxAddresses (a special class which uniquely specifies a particular Inbox on a particular virtual machine), using one of various methods described in the API documentation. When a message is sent, a separate copy of the message is sent to each InboxAddress to which the Outbox is bound at the time of sending. There is no reliability or ordering built into the Outbox module—therefore, there is no guarantee that all addressees of a message will receive it, nor that all addressees will receive multiple messages from the same Outbox in the same order.

An Inbox, like an Outbox, is assigned a unique name at construction. Inbox and Outbox names reside in different namespaces, so it is legal for an Inbox and an Outbox within the same virtual machine to have the same name. An Inbox, once created, receives all messages sent to it and places them in a first-in first-out queue. Any object within the Inbox's virtual machine which has a standard Java reference to an Inbox can remove the first message in the queue and retrieve information about that message (including its origin), at any time. It is not possible to examine the contents of a message in the queue without removing it.

Both Outbox and Inbox have integrated multicast support. Inbox has methods which allow individual Inboxes to join an arbitrary number of multicast groups (and thereby receive all messages sent to those groups). Outbox supports multicast groups transparently; since an Outbox can be bound to arbitrary addresses, binding an Outbox to a multicast address will result in messages being sent to the multicast group. An Outbox must be using the UDPDaemon as the bottom element of its protocol stack in order to send to multicast groups, since a TCP transmission to a multicast group is not possible.

4.4.2.2 Ordering

The classes OrderingSender and OrderingReceiver implement ordered messaging. Their function is to ensure that messages arrive at their destination in the order in which they are sent. This is done by tagging each message with a unique tag indicating its source, destination, and the message's sequence number.

4.4.2.3 Segmenting & Joining

The classes SegmentingSender and JoiningReceiver implement message segmenting. Since UDP datagrams have a maximum size, message segmenting is necessary to allow the transmission of large messages. The segmenting module does not require that segments be received in the order they were sent, but does require that all segments be received; it will not reconstruct partial messages.

4.4.2.4 Reliable Ordering

The classes ReliableOrderingSender and ReliableOrderingReceiver implement reliable ordered messaging. That is, they perform the functions of the classes OrderingSender and OrderingReceiver, and additionally ensure that all sent messages are successfully delivered.

The algorithm used by the reliable ordering module is a sliding window protocol with window size 1. It is, therefore, not an incredibly efficient way to ensure reliability. It is meant mainly as an example of how one might go about providing reliable messaging.

Because it is an acknowledgment-based algorithm, sliding window is inappropriate for multicast applications; the reliable ordering module, therefore, ignores any messages addressed to multicast groups, passing them through the stack with no additional processing.

4.4.2.5 Simple

The classes SimpleSender and SimpleReceiver are basically "no-operation" modules which do no message processing. They do, however, implement the basic functionality of sending-side and receiving-side modules, and are therefore useful as base classes for other protocol stack modules.

4.4.3 Other Support Classes

In addition to the wire protocol daemons and the protocol stack modules, there are several other support classes which play important roles in the ÜberNet system. This subsection briefly describes a few of the more important support classes.

4.4.3.1 Management

Most management functions of ÜberNet are encapsulated in the NetworkLayer class. This class performs all the tasks necessary for starting and shutting down the ÜberNet system, including the instantiation of the message router and the wire protocol daemons. The use of administration keys (class java.security.Key) prevents protocol stack modules and wire protocol daemons from making unauthorized calls to the management interface. In addition, the NetworkLayer class provides authentication routines so that custom ÜberNet components can verify the administration key, for use as part of administration tasks which do not affect the system as a whole.

4.4.3.2 Receiving-Side Message Routing

As discussed in Section 4.3, the ÜberNet system works by dynamically routing messages on the receiving side of a communication to the appropriate protocol stack modules. The object responsible for this routing is the ReceivingEventRouter. The ReceivingEventRouter examines the header data on incoming messages to determine which protocol stack modules to pass the message to for processing; if the appropriate module is not available, the ReceivingEventRouter downloads the class from a class repository (also specified in the message header data) and then uses it appropriately. For this reason, the ReceivingEventRouter is the most important component on the receiving side of communications in the ÜberNet system.

4.4.3.3 Class Loading

The loading of remote classes is carried out by a special class called the InfospheresClassLoader, which was developed for use in both ÜberNet and the Infospheres Infrastructure 2.0. The InfospheresClassLoader can load classes from web servers, FTP servers, and from locally-mounted file systems; it can work with individual bytecode files, ZIP files containing multiple bytecode files, and Java Archive (JAR) files. The communication streams which are used by ÜberNet to send objects across the network use the InfospheresClassLoader to obtain the bytecode for the objects' classes, which is also sent along with the objects so they can be properly instantiated at the remote location. A special interface, RemoteLoadable, allows the author of a class to specify a URL where the current bytecode for a class can always be obtained; this makes communication more efficient by allowing ÜberNet to send only the URL, instead of the entire object bytecode, when an object which implements the RemoteLoadable interface is communicated to another virtual machine.

Multiple instances of the InfospheresClassLoader are used by ÜberNet–the ReceivingEventRouter, for instance, uses a separate InfospheresClassLoader for each protocol module it loads remotely, both for security reasons and to avoid class namespace collisions. More information on remote class loading is available in both Sun's Java documentation and the ÜberNet User's Guide.

4.4.3.4 Resource Pools

Because Java has no concept of explicit memory allocation, it is a very easy language to use for complicated projects. However, the price for this is that many unnecessary object creations and memory allocations occur, which can sometimes cause serious performance degradation. To counter this, ÜberNet uses the concept of *resource pools*. Certain types of system resources, such as threads and streams, are allocated from pools and then returned to the pools for reuse when they are no longer needed. This reduces the overhead incurred by garbage collection and object creation, and as a result makes ÜberNet a more efficient and useful system.

Two such pooled resources which greatly improve the performance of ÜberNet are the subclasses of ObjectOutputStream and ObjectInputStream used within the system for object serialization. The pooling of these resources eliminates much of the overhead usually incurred by using Java's object serialization code, and allows ÜberNet to efficiently use serialized objects as messages rather than using dedicated message classes (as was done in version 1.0 of the Infospheres Infrastructure).

4.4.4 Extension Mechanisms

There are two main mechanisms for extending the functionality of the ÜberNet system. The first involves the addition of new protocol stack modules, and the second involves the addition of new wire protocol daemons.

To add a new protocol stack module to the ÜberNet system, all that needs to be done is to write the module and use it within a Java program. Any messages sent using that module will result in the correct module being automatically used on the receiving side, and the system is extended with no effort except for that of the programmer actually implementing the extension. For performance reasons, it may be desirable to install such extension classes on the local machines where messages will be received, but this is unnecessary for proper operation of the ÜberNet system.

Adding a new wire protocol daemon to the ÜberNet system is somewhat more complicated. New wire protocol daemons can be added both at runtime and before the system is started, but neither type of addition is as automatic as the addition of new protocol stack modules.

In order to add a wire protocol daemon at runtime, a special message must be sent to the ÜberNet system using a designated administrative mailbox. This causes the ÜberNet system to download the new daemon and its support classes, and instantiate the daemon on the local system. Because of the security issues involved, this capability of the system is inactive by default, and must be explicitly activated at ÜberNet startup time using the administrative interface.

In order to add a wire protocol daemon to the local system before starting ÜberNet the daemon and all its support classes must be placed in a special part of the local classpath, called the *extension path*, which is designated by means of a configuration file or a parameter passed to ÜberNet at startup. The ÜberNet system will search the extension path for wire protocol daemons, and instantiate all of them at system initialization time. Because this type of extension requires a local installation of classes, it does not have the same security problems as the runtime installation.

4.5 Performance Comparisons

A prerelease build of the ÜberNet implementation was compared with three other Java-based inter-object communication mechanisms, to determine its relative efficiency. The ÜberNet build being tested had not been optimized for communication speed, except insofar as it incorporated the resource pool mechanisms described in the previous section.

The comparison was based on the round-trip transfer time for messages of two different classes between two machines on the same Ethernet subnetwork. One class of messages was the standard Java class java.lang.Long (a 64-bit integer), and the other was a class containing an integer, a long integer, an array of 2048 random bytes, and an instance of the standard Java class java.util.Vector containing three instances of the standard Java class java.lang.String. Two implementations of this large message, one which implemented the java.io.Serializable interface and one which implemented ÜberNet's RemoteLoadable interface (described in Section 4.4.3.3), were tested with the ÜberNet protocols, while only the java.io.Serializable version was tested with the other protocols.

Three different sets of protocols within ÜberNet–reliable ordered UDP on both outgoing and incoming messages, TCP on both outgoing and incoming messages, and reliable ordered UDP on outgoing messages with TCP on incoming messages—were tested. The three other inter-object communication mechanisms used were straight TCP sockets, the info.net package from version 1.0 of the Infospheres Infrastructure, and the RMI-based network layer from an internal prerelease build of version 2.0 of the Infospheres Infrastructure. With the TCP and UDP sockets, the messages actually being communicated over the wire consisted only of the serialized object; in the case of the Infospheres Infrastructure-derived communication mechanisms, the Infospheres Infrastructure itself determined the format of the data being communicated over the wire. The machines used for the test were 167MHz UltraSPARCs with

Protocol	Run 1 Mean	Run 1 SD	Run 2 Mean	Run 2 SD	Run 3 Mean	Run 3 SD	Run 4 Mean	Run 4 SD	Run 5 Mean	Run 5 SD
ÜberNet (TCP)	53	14	53	14	53	14	53	14	54	16
ÜberNet (UDP)	53	20	52	22	53	20	53	21	53	21
ÜberNet (Hybrid)	54	14	54	15	54	16	55	17	54	15
TCP Sockets	5	2	5	3	5	3	5	3	5	3
info.net	29	6	29	7	29	6	40	185	34	157
Infospheres 2	11	3	11	4	11	4	11	7	11	6

Table 4.1: Average round-trip message transfer times and standard deviations for individual runs of 1000 java.lang.Long messages. All values are in milliseconds.

Protocol	Run 1 Mean	Run 1 SD	Run 2 Mean	Run 2 SD	Run 3 Mean	Run 3 SD	Run 4 Mean	Run 4 SD	Run 5 Mean	Run 5 SD
ÜberNet (TCP)	74	23	73	21	74	19	74	20	74	21
ÜberNet (UDP)	95	160	133	450	104	225	106	224	90	18
ÜberNet (Hybrid)	90	160	86	26	95	226	88	160	86	23
TCP Sockets	20	16	20	15	20	16	20	15	20	18
info.net	29	7	29	10	30	7	29	9	29	5
Infospheres 2	26	8	26	6	26	6	27	9	26	6

Table 4.2: Average round-trip message transfer times and standard deviations for individual runs of 1000 large messages (java.io.Serializable implementation). All values are in milliseconds.

256MB of physical memory, running Sun's Java Development Kit version 1.1.5.

For each test, 5 sets of 1000 messages each were sent round-trip between the machines. When each message returned to the sender, its transit time (in milliseconds) was calculated and recorded before the next message was sent. The 1000 transit times were then used to obtain the mean transit time and the standard deviation of the transit times for each run. The results of these tests are summarized in Tables 4.1 through 4.4, and histograms of the transfer times for all 15 sets of 5000 messages can be found in Appendix A.

As the tables show, the performance of the three other inter-object communication mechanisms was superior to that of ÜberNet by a fairly large margin—at best, ÜberNet is slower by almost a factor of 4; at worst, by more than a factor of 10. However, this is not unacceptable performance, for two main reasons.

Protocol	Run 1 Mean	Run 1 SD	Run 2 Mean	Run 2 SD	Run 3 Mean	Run 3 SD	Run 4 Mean	Run 4 SD	Run 5 Mean	Run 5 SD
ÜberNet (TCP)	71	17	70	17	71	18	71	20	71	16
ÜberNet (UDP)	89	17	91	25	98	52	89	20	89	19
ÜberNet (Hybrid)	81	21	81	18	82	19	82	21	81	17

Table 4.3: Average round-trip message transfer times and standard deviations for individual runs of 1000 large messages (RemoteLoadable implementation). All values are in milliseconds.

Protocol	java.lang.Long Mean	SD	Large (Serializable) Mean	SD	Large (RemoteLoadable) Mean	SD
ÜberNet (TCP)	53	15	74	21	71	18
ÜberNet (UDP)	53	21	106	21	91	30
ÜberNet (Hybrid)	54	16	90	144	81	19
TCP Sockets	5	3	20	16	–	–
info.net	32	109	30	8	–	–
Infospheres 2	11	5	26	7	–	–

Table 4.4: Average round-trip message transfer times and standard deviations over all runs. All values are in milliseconds.

First, ÜberNet is far more flexible than any of the other systems tested, and such flexibility naturally results in extra communications overhead—as an example, none of the other 3 systems can properly handle the case where the bytecode for an object communicated to a different virtual machine does not exist at its destination, but ÜberNet does handle this situation. Second, the version of ÜberNet which was tested was implemented primarily as a proof of concept for a dynamic workflow-based communication system, and was never optimized for speed. We do believe that ÜberNet's speed can be improved significantly, both by making internal algorithmic improvements to ÜberNet itself and by taking advantage of advancements in Java Virtual Machine technology which will improve the performance of highly multithreaded applications such as the ÜberNet system; we will focus more on efficiency issues when implementing the next version of ÜberNet.

Chapter 5

Conclusion

In this thesis, we have presented the concept of dynamic distributed workflow and identified several interesting implementation and theoretical questions associated with it. We then presented ÜberNet, an inter-object communication system implemented in Java which was designed as a limited dynamic workflow system, and described its implementation and operational details. In ÜberNet, every message is a job with its own workflow schema, and every job carries its schema with it as it travels through the workflow system. The construction of the ÜberNet system is a large step toward the construction of more general dynamic distributed workflow systems, both because it implements a subset of the functionality of such a system and because it provides a robust inter-object communication mechanism upon which to build such a system.

Future research and implementation work will be focused on two main areas. First, we will improve and extend the ÜberNet system; we intend to extend the dynamic workflow capabilities of the system to include non-linear workflow constructs, as well as to improve the system's efficiency both by improving internal algorithms and by taking advantage of new language features introduced in the next version of the Java language. By extending and improving ÜberNet in this fashion, we will create an even more reliable and robust communication infrastructure on which to base future dynamic distributed workflow implementations. Second, we will explore in depth the implementation and theoretical questions discussed in Chapter 2; we will try to determine what type of specification is appropriate for dynamic distributed workflow systems, and also examine the various reasoning methods which we apply to distributed systems today to determine which of these are applicable to dynamic distributed workflow systems and where new reasoning methods and constructs are needed.

Appendix A

Histograms of Performance Tests

The histograms contained in this appendix show the distribution of round-trip message times for the fifteen performance tests described in Chapter 4. The axes on all the histograms are identical for comparison purposes; for most of the tests, this results in some outlying data points not being represented in the histograms. In the most extreme case, that of ÜberNet UDP with large messages implementing the RemoteLoadable interface, 76 data points (approximately 1.5% of the data) are not represented in the histogram. No data points were left out while calculating the statistics which appear in Section 4.5.

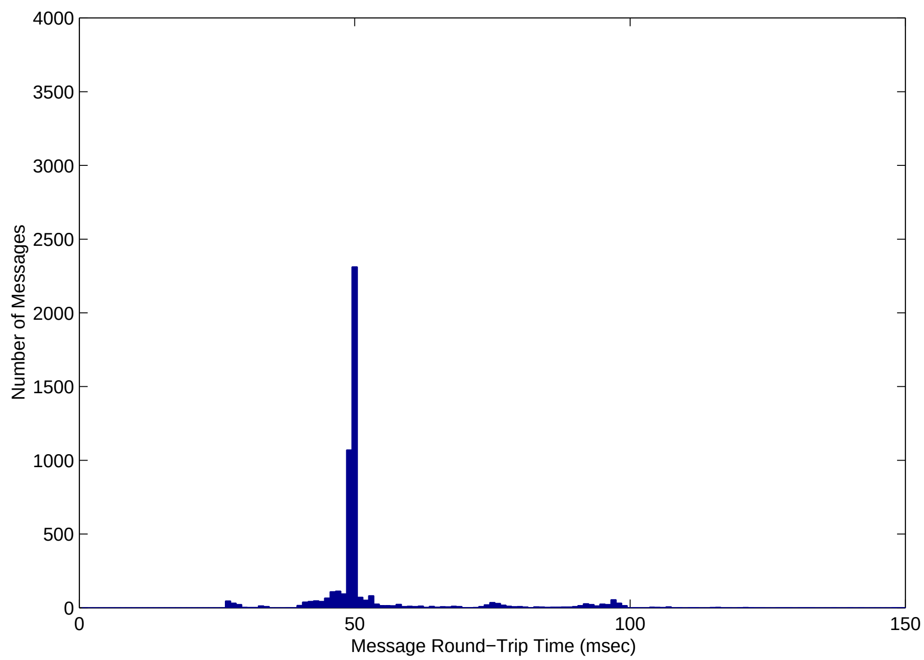

Figure A.1: Histogram of ÜberNet TCP performance test with 5000 messages of class java.lang.Long. 6 data points (approximately 0.1% of the data) are not represented in the histogram.

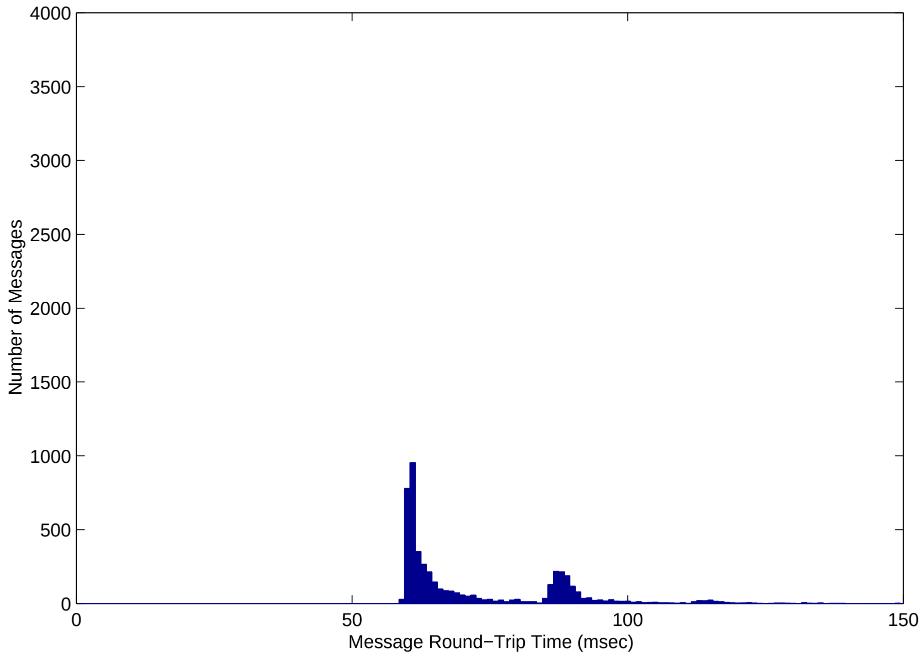

Figure A.2: Histogram of ÜberNet TCP performance test with 5000 large messages implementing the java.io.Serializable interface. 37 data points (approximately 0.7% of the data) are not represented in the histogram.

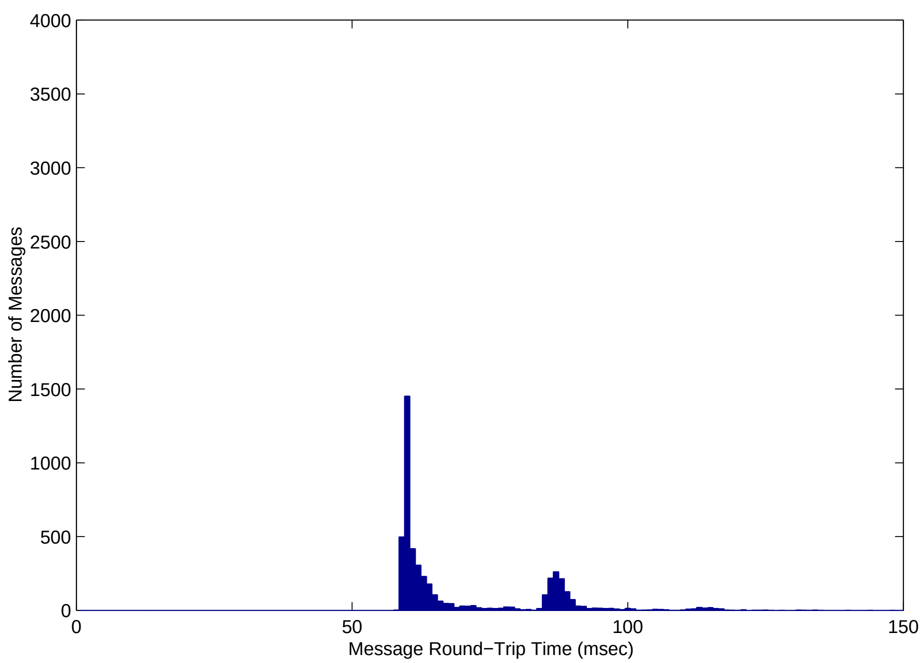

Figure A.3: Histogram of ÜberNet TCP performance test with 5000 large messages implementing the RemoteLoadable interface. 18 data points (approximately 0.4% of the data) are not represented in the histogram.

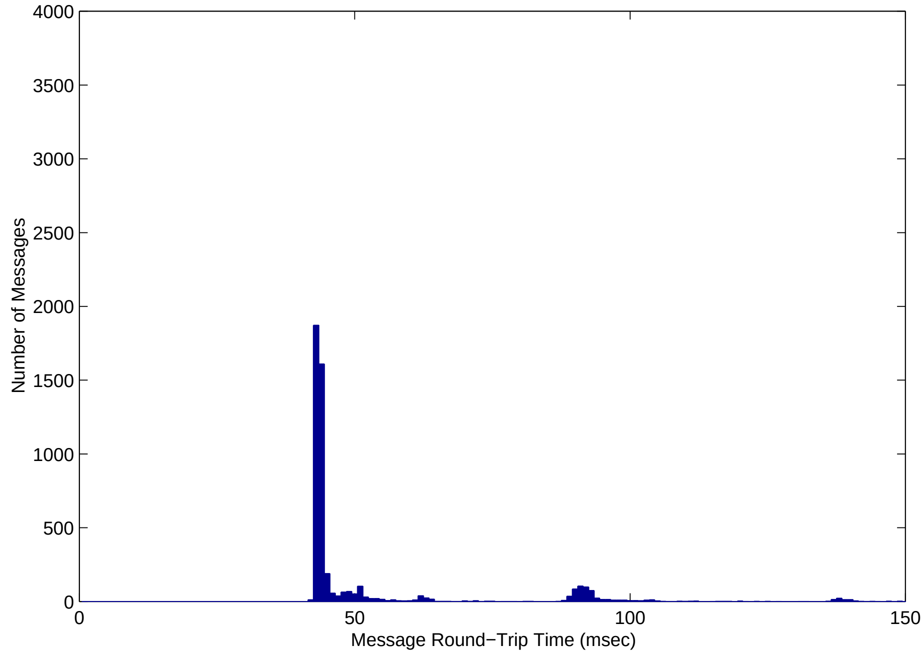

Figure A.4: Histogram of ÜberNet UDP performance test with 5000 messages of class java.lang.Long. 13 data points (approximately 0.3% of the data) are not represented in the histogram.

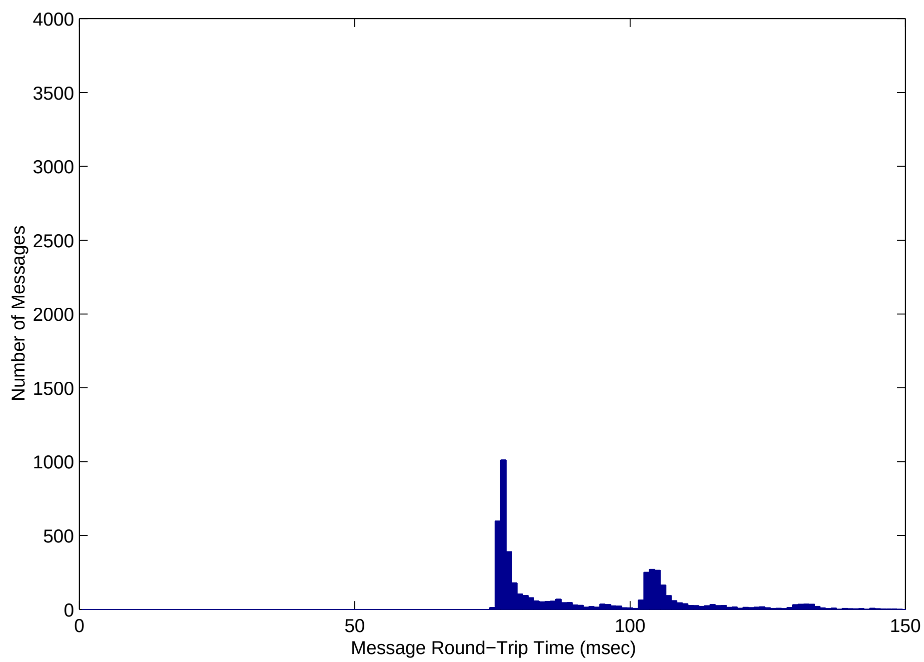

Figure A.5: Histogram of ÜberNet UDP performance test with 5000 large messages implementing the java.io.Serializable interface. 68 data points (approximately 1.4% of the data) are not represented in the histogram.

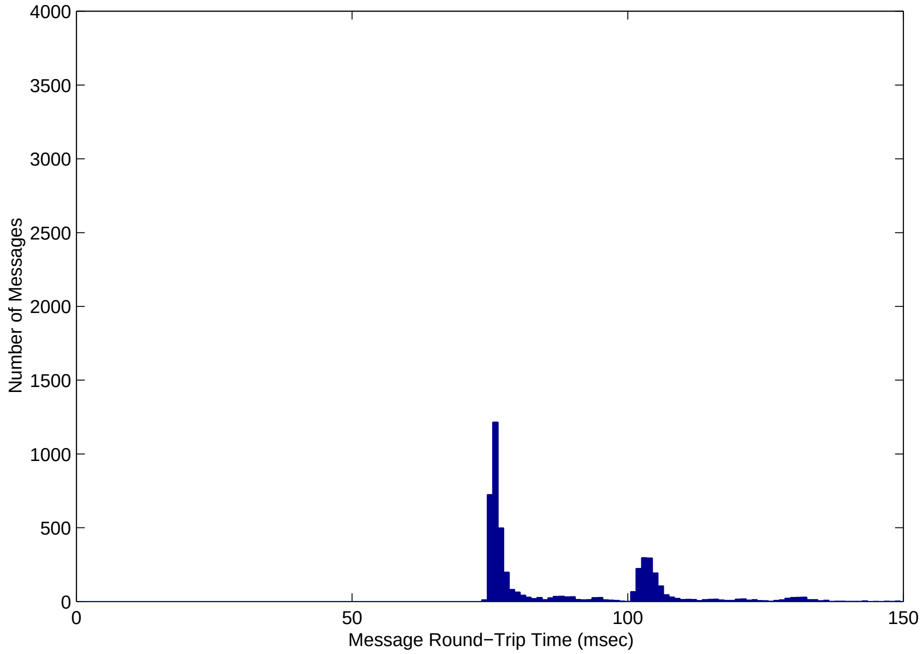

Figure A.6: Histogram of ÜberNet UDP performance test with 5000 large messages implementing the RemoteLoadable interface. 76 data points (approximately 1.5% of the data) are not represented in the histogram.

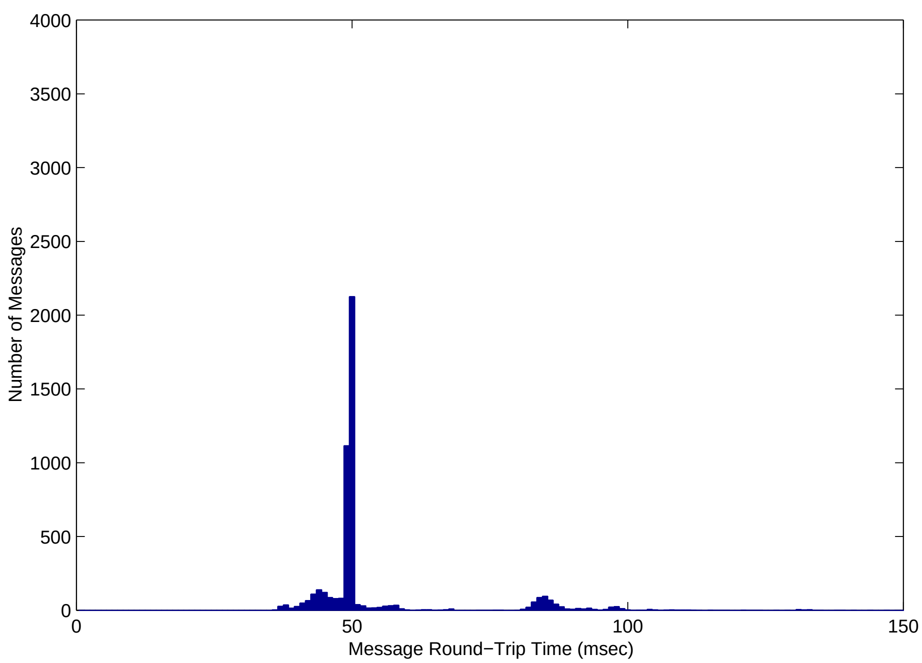

Figure A.7: Histogram of ÜberNet Hybrid performance test with 5000 messages of class java.lang.Long. 6 data points (approximately 0.1% of the data) are not represented in the histogram.

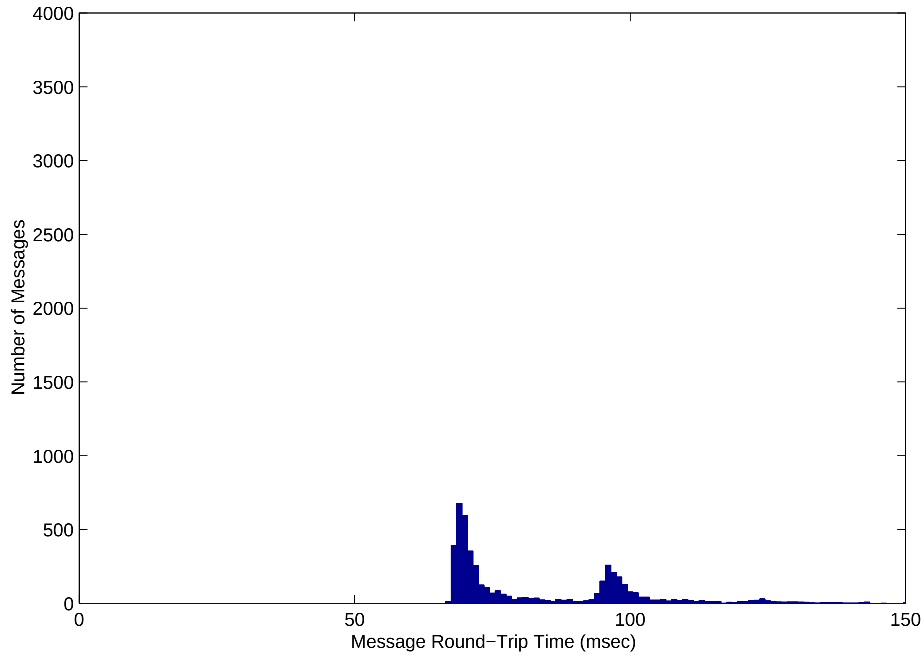

Figure A.8: Histogram of ÜberNet Hybrid performance test with 5000 large messages implementing the java.io.Serializable interface. 70 data points (approximately 1.4% of the data) are not represented in the histogram.

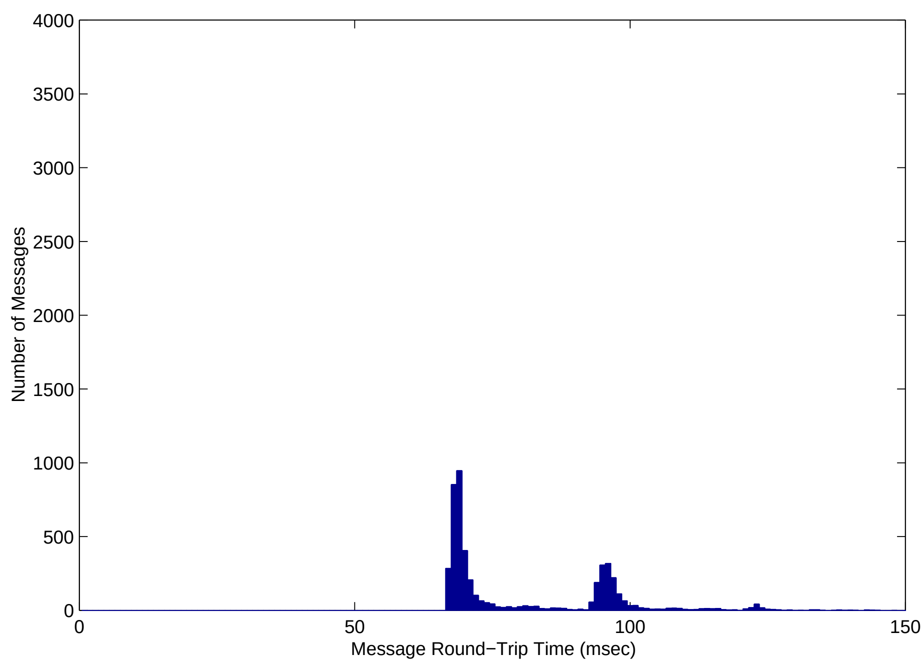

Figure A.9: Histogram of ÜberNet Hybrid performance test with 5000 large messages implementing the RemoteLoadable interface. 26 data points (approximately 0.5% of the data) are not represented in the histogram.

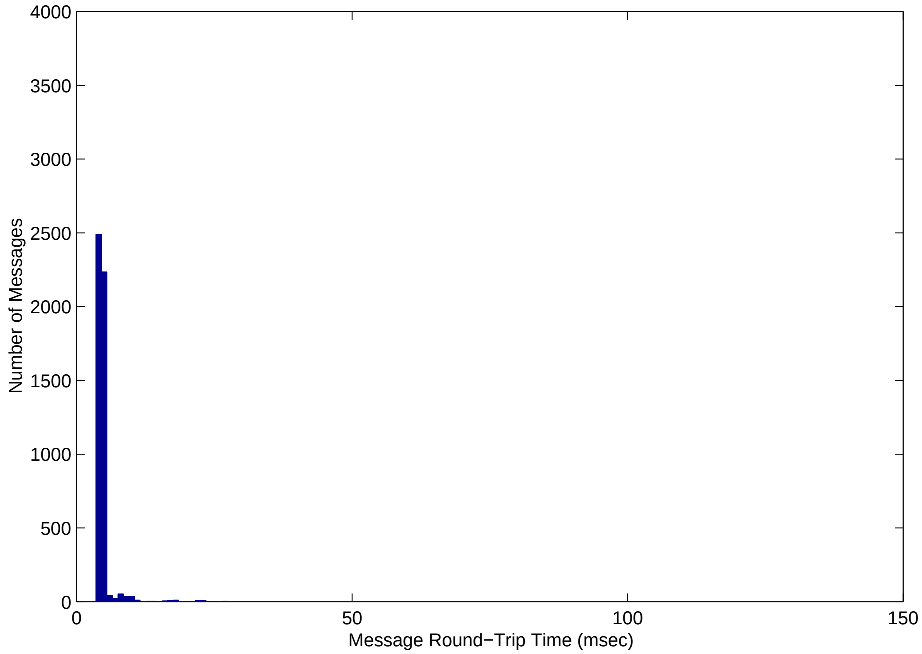

Figure A.10: Histogram of TCP Sockets performance test with 5000 messages of class java.lang.Long.

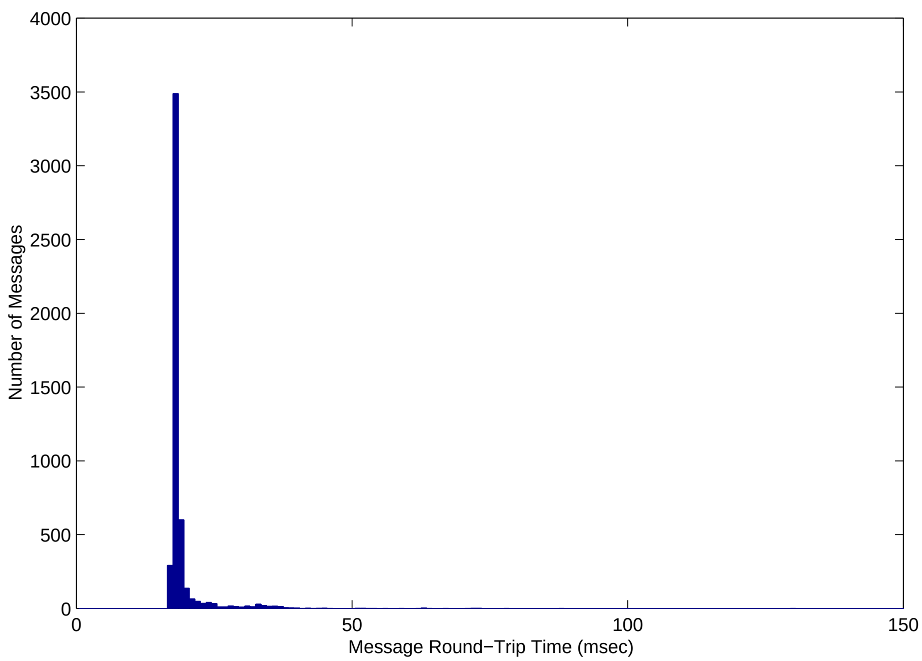

Figure A.11: Histogram of TCP Sockets performance test with 5000 large messages implementing the java.io.Serializable interface. 11 data points (approximately 0.2% of the data) are not represented in the histogram.

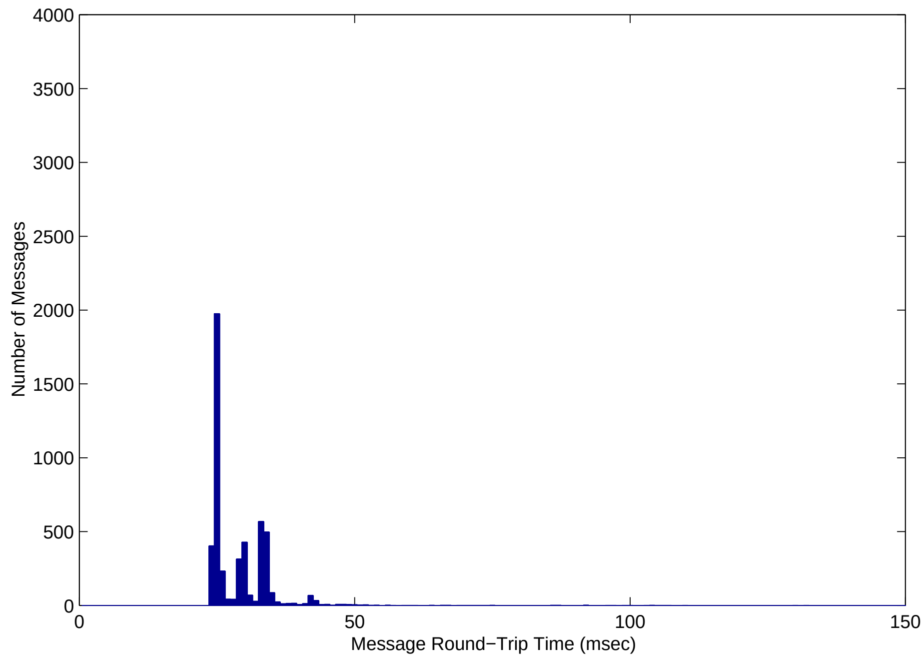

Figure A.12: Histogram of info.net performance test with 5000 messages of class java.lang.Long. 7 data points (approximately 0.1% of the data) are not represented in the histogram.

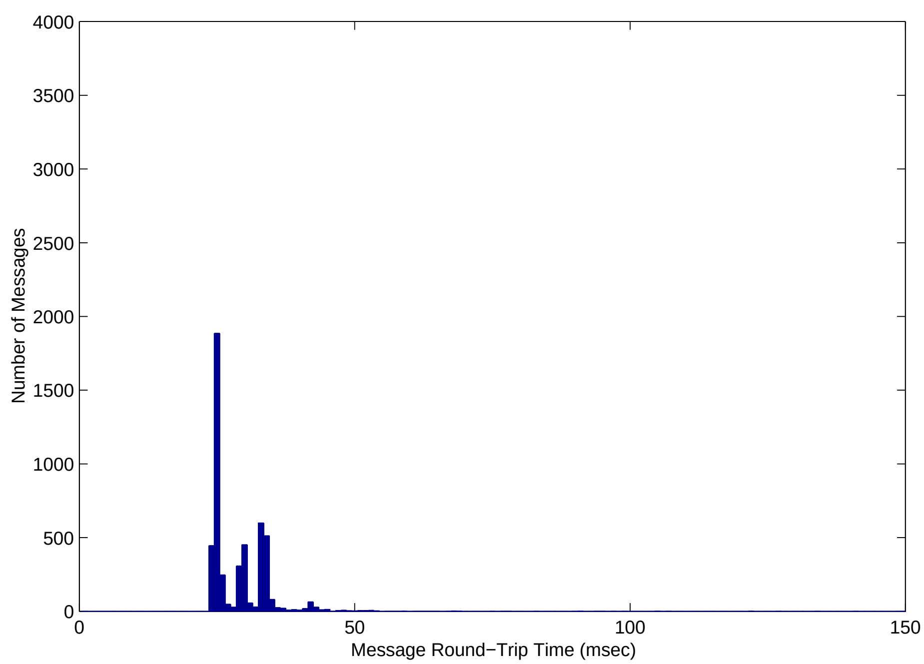

Figure A.13: Histogram of info.net performance test with 5000 large messages implementing the java.io.Serializable interface. 2 data points (0.04% of the data) are not represented in the histogram.

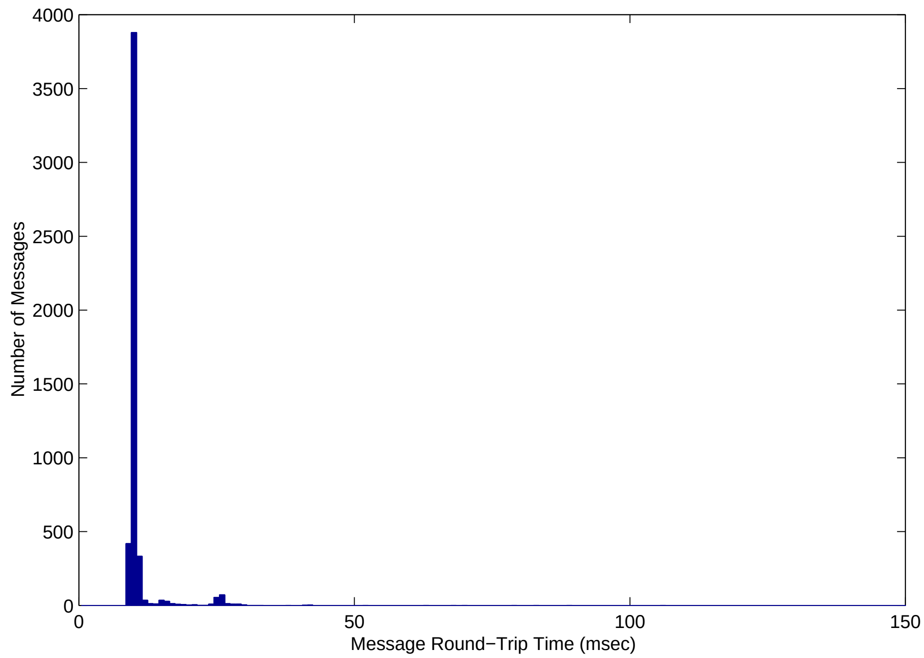

Figure A.14: Histogram of Infospheres 2 (RMI) performance test with 5000 messages of class java.lang.Long. 1 data point (0.02% of the data) is not represented in the histogram.

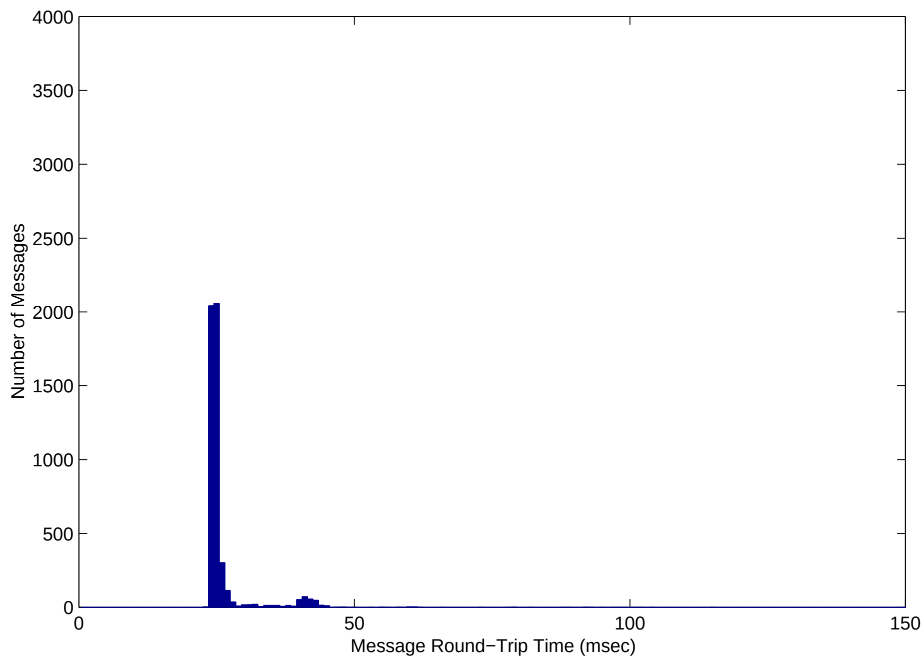

Figure A.15: Histogram of Infospheres 2 (RMI) performance test with 5000 large messages implementing the java.io.Serializable interface. 2 data points (0.04% of the data) are not represented in the histogram.

Bibliography

[1] R. Andry. 'Exploiting Transaction and Object Semantics to Increase Concurrency'. Applications in Parallel and Distributed Computing, 1994.

[2] F. Casati, S. Ceri, B. Pernici and G. Pozzi. 'Workflow Evolution'. Data and Knowledge Engineering Volume 24, Issue 3, 1998.

[3] M. Erdos, B. Hartman, M. Mueller. 'Security Reference Model for the Java Developer's Kit 1.0.2'. Sun Microsystems Technical Report, 1996.

[4] The Infospheres Research Group. 'The Infospheres Infrastructure User's Guide'. Technical Report, California Institute of Technology, 1997.

[5] The Infospheres Research Group. 'The Infospheres Infrastructure 2.0 User's Guide'. Technical Report, California Institute of Technology. 1998.

[6] JavaSoft, Inc. 'Java Cryptography Architecture: API Specification and Reference'. 1997.

[7] M. Merz, B. Liberman and W. Lamersdorf. 'Using Mobile Agents to Support Interorganizational Workflow Management'. Applied Artificial Intelligence, November 1997.

[8] Object Management Group. *CORBA/IIOP 2.2 Specification*. 1998.

[9] Sun Microsystems. 'The Java Beans Specification'. Version 1.01, 1997.

[10] Sun Microsystems. 'Java Studio'. *http://www.sun.com/studio/*, 1998.

[11] The Workflow Management Coalition. 'The Workflow Reference Model'. Document Number WFMC-TC-1003, 1994.

[12] The Workflow Management Coalition. 'Terminology and Glossary'. Document Number WFMC-TC-1011, 1996.

[13] D. Zimmerman. 'ÜberNet: The Infospheres Network Layer - User's Guide'. Technical Report, California Institute of Technology, 1998.