A Framework for Structured Distributed Object Computing

K. Mani Chandy, Joseph Kiniry, Adam Rifkin, Daniel Zimmerman,
Wesley Tanaka, and Luke Weisman*
infospheres@cs.caltech.edu

Computer Science 256-80
California Institute of Technology
Pasadena, California 91125

http://www.infospheres.caltech.edu/

February 7, 1997

Abstract

This paper presents a four-faceted framework for distributed applications that use world-wide networks connecting large numbers of people, software tools, monitoring instruments, and control devices. We describe a class of applications, identify requirements for a framework that supports these applications, and propose a design fulfilling those requirements. We discuss some initial experiences using the framework, and compare our design with other approaches.

1 Personal Command and Control Applications

The global information infrastructure will soon connect large numbers of processes that manage devices and human interfaces. Interprocess communication will allow processes to respond to events on such devices as medical monitoring equipment, scientific instruments, home appliances, and security systems, and on such software as scheduling programs, document management systems, Web browsers, and complex computation engines.

The contribution of this paper is a simple, generic framework for developing distributed systems for personal applications. By employing our framework, developers can quickly build interactive command and control processes that run over the Internet. Our framework is composed of four facets: (i) *processes* are persistent communicating objects; (ii) *personal networks* provide wiring diagrams and behaviors for these connected processes; (iii) *sessions* are transactions performed by

*The Caltech Infospheres Project is sponsored 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 the Advanced Research Projects Agency, and by Novell, Inc.

the processes participating in a personal network; and (iv) *infospheres* are custom collections of processes for use in personal networks.

Infospheres and Personal Networks. *Warfighter's infosphere* is a term coined by the military to represent the electronic interface between a military unit and its environment. This human-computer interface is provided by the military *C4I* (command, control, communications, computers, and intelligence) infrastructure.

Our goal is to provide a framework for transitioning the concepts of infospheres and C4I to individuals and small organizations to create analogous lightweight command and control systems. *Personal networks* are roadmaps for such systems, specifying processes arranged in a topology, with a specified cooperative behavior. For example, a person in Nevada may have a *emergency notification personal network* that incorporates processes for medical monitoring devices in her parents' home in Florida, security and utility systems in her home and car in Reno, a global position sensing device on her teenage son's car in Montréal, a Nikkei Market stock ticker tape, and software programs that monitor urgent pages and e-mails.

Task Forces for Organizations. Personal networks can also be used by institutions and businesses to create task forces to handle short-term situations. The structure of personal networks comprises the organizational, informational, and workflow structures of the corresponding task force. *Workflow* describes the manner in which jobs are processed in stages by different processes [22].

One example of a task force is a panel that reviews proposals submitted to the National Science Foundation (NSF). Panel members come from a variety of institutions, and the panel has an organizational structure with a general chair, subcommittees, primary reviewers, and secondary reviewers. The panel's informational structure includes the hierarchy of proposals and reviews, and the panel's workflow is the flow of proposal and review copies. The panel has its own organizational, informational, and workflow structures that coexist with those of NSF. In this sense, NSF's organizational and informational structures adapt in a dynamic, but systematic, way to include new people and resources as needed.

2 Requirements Analysis

A framework to support personal networks (and their components) should satisfy four main requirements: scalability, simplicity, security, and adaptability.

Scalability. Personal networks should scale to include devices, tools, and people connected to the Internet. The critical scaling issue is not the number of processes connected in a personal network, but rather the size of the pool from which the processes in personal networks are drawn. The only limit to the number of processes connected in a personal network is the number of activities that can be managed effectively. However, issues of scaling in naming, connections, and services depend on the size of the global set of processes and resources.

Personal networks should be tolerant of wide ranges of quality of service because the processes in a personal network can exist on a single system or span several continents. The framework should both support large numbers of concurrent personal networks and provide a core set of services for creating and using personal networks.

Simplicity. The usage and programming model for personal networks should be simple enough to be usable by anyone. The simplicity of dialing a telephone led to the widespread use of telephones despite the complexity of the worldwide telecommunications network. If personal networks are to become effective tools, their use should be similarly intuitive. So, the model's API should be easy for programmers to learn quickly, and the accompanying visual tools should allow non-programmers to use palettes of existing constructs to customize their personal networks.

Security. A research instrument shared by several people may have one interface for setting control parameters and a different interface, accessible by a small set of authorized personnel, for accessing the data recorded by the instrument. Also, instruction messages sent to the "modify-parameter" interface may be of a different type than instructions to the "read-data" interface. Therefore, the framework should allow processes to have multiple typed interfaces and provide the ability to set security restrictions on at least a per-interface basis.

Adaptability. It should be possible to create and modify personal networks rapidly and flexibly, because task forces often need to be set up quickly and in an ad hoc manner. Network topologies should be emergent rather than static, so processes should be able to create and delete connections during a session. Additionally, personal network processes should be able to communicate with applications and devices that were unknown or nonexistent prior to the creation of the personal network. So, the framework should be extensible enough to support interoperability with other distributed technologies.

3 Design of an Extensible Framework

Our framework employs three structuring mechanisms: personal networks, to facilitate long-term collaborations between people or groups; sessions, to provide a mechanism for carrying out the short-term tasks necessary within these personal networks; and infospheres, to allow customization of processes and personal networks.

To illustrate these structuring mechanisms, consider a consortium of institutions carrying out research on a common problem. It has a personal network composed of processes that belong to the infospheres of the consortium members. This personal network is a structured way to manage the collection of resources, processes, and communication channels used in distributed tasks such as simulating financial scenarios, determining meeting times, and querying distributed databases. Each session of this personal network handles the acquisition, use, and release of resources, processes, and channels for the life of a specific task.

Infospheres are discussed in our framework user's guide [12]. This paper focuses on the conceptual models for processes, personal networks, and sessions.

3.1 Conceptual Model: Processes

Processes are the persistent communicating objects that manage devices and interfaces. In our framework, we call these processes *djinns*.

Process States. A given process can be in one of three states. An *active* process is a process that has at least one executing thread; it can change its state and perform any tasks it has pending, including communications. A *waiting* process has no executing threads; its state remains unchanged while it is waiting, and it remains in the waiting state until one of a specified set of input ports becomes nonempty, at which point it becomes active and resumes execution. Active and waiting processes are collectively referred to as a *ready* process.

Ready processes occupy process slots and can make use of other resources provided by the operating system. By contrast, processes in the third state, *frozen*, do not occupy process slots. In fact, frozen processes do not use any operating system resources except for the persistent storage, such as a file or a database, that is used to maintain process state information.

Freezing, Summoning, and Thawing Processes. Associated with each process is a *freeze* method, that saves the state of the process to a persistent store, and a *thaw* method, that restores the process state from the store. Typical processes remain in the frozen state nearly all the time, and therefore require minimal resources. In our framework, only a waiting process can be frozen, and it can only be frozen at process-specified points. When its freeze method is invoked, a process yields all the system resources it holds.

A ready process can *summon* a frozen process. The act of summoning instantiates the frozen process, causes its thaw method to be invoked, and initiates a transition to the ready state. If a process is ready when it is summoned, it remains ready. In either case, a summoned process remains ready until it receives at least one message from its summoner or a specified timeout interval elapses.

Mobile Processes. Frozen processes can move from one machine to another, but ready processes cannot. This restriction allows ready processes to communicate using our framework's underlying fast transport layer, that requires unchanging addresses for communication resources. All processes have a permanent "home address" from which summons can be forwarded. Once a process becomes ready at a given location, it remains at that location until the process is next frozen. The persistent state of a process is always stored at the home address of that process.

3.2 Conceptual Model: Personal Networks

Conceptually, a personal network is a wiring diagram, analogous to a home entertainment system, with directed wires connecting device outputs to the inputs of other devices. We chose this model for its simplicity [3]. A personal network consists of an arrangement of processes and a set of directed, typed, secure communication channels connecting process output ports to the input ports of other processes; its topology can be represented by a labeled directed graph. Note that, unlike home entertainment system components, processes can freely create input ports, create output ports, and change wire connections.

Communication Structures. Processes communicate with each other by passing messages. Associated with each process is a set of *inboxes* and a set of *outboxes*. Inboxes and outboxes are collectively called *mailboxes*. Every mailbox has a type and an access control list, both of which are used to enforce personal network structure and security. These mailboxes correspond to the device inputs and outputs used in the wiring diagram conceptual model.

Process interconnections are asymmetric; a process can connect any of its outboxes to any set of inboxes for which it has references. A connection is a first-in-first-out, directed, secure, error-free broadcast channel from the outbox to each connected inbox. Our framework contains support for message prioritization, available through standard multithreading techniques.

Message Delivery. Our framework communication layer works by removing the message at the head of a nonempty outbox and appending a copy to each connected inbox. If the communication layer cannot deliver a message, an exception is raised in the sender containing the message, destination inbox, and specific error condition. Our system uses a sliding window protocol [17] to manage the messages in transit.

Every message at the head of an outbox will eventually be handled by the communication layer. The conceptual model uses asynchronous messages rather than remote procedure calls, to be tolerant of the range of message delays experienced along different links of the Internet. As a result, we can think about message delivery from an outbox to inboxes as a simple synchronous operation even though the actual implementation is asynchronous and complex.

Dynamic Structures. A process can create, delete, and change mailboxes. The operation of creating a mailbox returns a global reference to that mailbox. This reference can then be passed, in messages, to other processes. Since a process can change its connections and mailboxes, the topology of a personal network can evolve over time as required to perform new tasks.

As long as a process remains ready, references to its mailboxes are valid; when a process is frozen, all references to its mailboxes become invalid. Since all references to the mailboxes of frozen processes are invalid, frozen processes can move and then be thawed, at which point the references to their mailboxes need be refreshed via a summons. Because no valid references to their mailboxes exist, frozen processes cannot participate in sessions.

3.3 Conceptual Model: Sessions

Operationally, a session is a task carried out by (the processes in) a personal network [4]. It is *initiated* by a process in the personal network, and is *completed* when the task has been accomplished. A later session may use the same processes to carry out another task. Thus, a personal network consists of a group of processes in a specified topology, interacting in sessions to perform tasks.

The Session Constraint. We adopt the convention that sessions must satisfy the two part *session constraint*:

1. As long as any process within the session holds a reference to a mailbox belonging to another process within the session, that reference must remain valid.
2. A mailbox's access control list cannot be constricted as long as any other process in the session holds a reference to that mailbox.

The session constraint ensures that, during a session, information flows correctly between processes.

A session is usually started by the process initially charged with accomplishing a task. This *initiator process* creates a session by summoning the processes that will initially participate. It then obtains references to their mailboxes, passes these references to the other processes, and makes the appropriate connections of its outboxes to the inboxes of the participating processes. We discuss session implementation and reasoning issues in Section 4.

There are many ways of satisfying the session constraint. One simple way is to ensure that once a process participates in a session it remains ready until the session terminates, and that once a process sends its mailbox references to other processes it leaves these mailboxes unchanged for the duration of the session. Another approach is to have the initiating process detect the completion of the task through a diffusing computation, after which it can inform the other session members that the session can be disbanded.

An Example Session. An example of a session is the task of determining an acceptable meeting time and place for a quorum of committee members. Each committee member has an infosphere containing a calendar process that manages his or her appointments. A personal network describes the topology of these calendar processes. A session initiator sets up the network connections of this personal network. The processes negotiate to find an acceptable meeting time or to determine that no suitable time exists. The task completes, the session ends, and the processes freeze. Note that the framework does not *require* that processes freeze when the session terminates.

During a session, the processes must receive the quality of service they need to accomplish their task. Therefore, communication is routed directly from process to process, rather than through object request brokers or intermediate processes as in client-server systems. Once a session is constructed, our framework's only communication role is to choose the appropriate protocols and channels. A session can negotiate with the underlying communication layer to use the most appropriate process-to-process mechanism. The current framework supports only UDP, but we plan in future releases to support a range of protocols such as TCP and communication layers such as Globus [6].

4 Structuring Mechanisms

Personal networks and sessions can be used not only as structuring mechanisms, but also for reasoning about the services provided to distributed systems.

4.1 Reasoning About Sessions

Consider a consortium of institutions working together on a research project. From time to time, people and resources of the consortium carry out a collaborative task by initiating a session, setting up connections using the personal network, performing the necessary machinations for the task, disbanding the connections, and terminating the session. Furthermore, several sessions initiated by the same consortium may be executing at the same time. For instance, a session to determine a meeting time for the executive committee and a session that reads measurements from devices in order to carry out a distributed computation could execute simultaneously. Moreover, the same process may participate concurrently in sessions initiated by different consortia or task forces. For example, a calendar manager may participate concurrently in sessions determining meeting times for a scout troop and a conference program committee. Our framework allows processes to participate in concurrent sessions [4].

A resource may be requested by a session in either *exclusive mode* or *nonexclusive mode*. For example, a visualization engine may need to be in exclusive mode for a task: while the task is executing, no other task can access it. However, a process managing a calendar can be useful in nonexclusive mode: several sessions can not only read the calendar concurrently, but also modify different parts of the calendar concurrently.

Because we cannot predict a priori the applications and sessions that will run concurrently, we restrict access to modify the states of the processes participating in a given session, to reason about that session's behavior. Such restrictions are currently provided in thread libraries by mutexes and monitors; our challenge is to provide similar constructs with our framework for use in distributed systems in a generic, extensible, and scalable manner.

4.2 Services for Sessions

New capabilities are added to our framework either by subclassing existing processes or by extending the framework. A *service* is a framework extension that is applicable to an assortment of distributed algorithms. Examples include mechanisms for locking, deadlock avoidance, termination detection, and resource reservation.

Locking Mechanisms. Even if a process participates concurrently in several sessions, there are points in a computation when one session needs exclusive access to certain objects. For example, at some point, the session determining the meeting time for a program committee needs to obtain exclusive access to the relevant portions of the calendars of all the committee members. Therefore, one service our framework should provide is the acquisition of locks on distributed objects accessed during a session. A great deal of work exists relating to locking in distributed databases and distributed transaction systems [9, 15]. Presently, our framework provides only an exclusive lock on an object, but the framework can be extended to include other types of locks, such as read and write locks.

Deadlock Avoidance. If sessions lock objects in an incremental fashion, deadlock can occur. For instance, if one session locks object A and then object B, and another session locks B and then

A, the sessions may deadlock because each session holds one object while waiting for the other. Therefore, our framework deals only with the case where a session requests locks on a set of objects only when it holds no locks; a session must release all locks that it holds before requesting locks on a different set of processes. An alternative solution would be to allow incremental locking in some total ordering, but we are not exploring this solution because it does not scale to distributed systems drawn from a worldwide pool of objects.

Termination Detection and Resource Reservation. Other services that can be extended into our framework include session termination detection and resource reservation. Termination detection can be used by an initiating process of a session to, for instance, determine when the states of the processes involved in the session need to be "rolled back" in the event of a failure. Resource reservation is a generic service through which the resources required by a session can be reserved for some time in the future. For instance, one might reserve the visualization engine at location X and the monitoring instrument at location Y for the earliest time after 5:00 PM today.

5 Experience With Our Framework

Two examples illustrate the ease with which programmers have used our framework to develop distributed systems. Using our model and middleware packages, a programmer was able to specify, design, reason about, and implement a distributed calendar application in under a week. Since our infrastructure handled the communication layer, the programmer could concentrate his skills on the high-level design and implementation.

Also, using our framework, given a specification of the processes and communication protocols, the students in an undergraduate class at Caltech were able to write processes that participated in a five-card draw poker tournament session. The students were given a week to design, reason about, and implement their poker-playing processes; we spent approximately the same amount of time specifying those processes and their interactions.

In both these cases, patterns helped the programmers develop their code quickly. Patterns encapsulate software solutions to common problems [8], and our framework has incorporated some applications of concurrency patterns in Java [14]. Initial experience with our framework has suggested several other patterns, both for collaborations between processes and for state-transition systems.

Collaboration Patterns. Several patterns of collaboration network topologies have emerged from our exploration of personal networks. A personal network consisting of a "master" process maintaining all modifications to an object shared by the other objects of the personal network fits the *Personal Network Star* pattern. For example, a concurrent document editing system with a single process responsible for maintaining changes during a personal network would match this pattern. This pattern roughly corresponds to a system with a single server with a set of clients, though more sophisticated systems (such as a hierarchy with multiple servers and multiple clients) could also be developed. The Personal Network Star pattern was employed in both the calendar and poker applications mentioned above.

A personal network in which each of the processes collaborate without a master, with all modifications announced to the entire group, fits the *Personal Network Full Connection* pattern. For example, a concurrent document editing system in which every process sends every modification to every other process, and every process is responsible for updating the local view of the shared object, would match this pattern. This pattern roughly corresponds to a peer-to-peer distributed system, though more sophisticated systems (such as different priorities for different peers) could also be developed.

A personal network in which messages are propagated in a ring during collaboration fits the *Personal Network Ring* pattern. For example, a document editing system in which the session-initiator process has a document and makes changes to it, then sends the modified document to the next process for it to make changes, and so on until the document is returned to the session-initiator process, would match this pattern. This pattern roughly corresponds to a workflow distributed system, though more elaborate workflow templates could also be developed. The Personal Network Full Connection and Personal Network Ring patterns were used in the poker applications mentioned above.

We are investigating other middleware patterns as well, such as hierarchical broadcast using publishing and subscribing processes, and dataflow using waiting and notification processes.

State-Transition System Patterns. In addition to collaboration patterns among the processes in a personal network, our experiences with user interfaces for describing network topologies has given rise to a pair of state-transition system patterns. Using these patterns, developers can design and reason about the changes of state in the processes participating in a session.

One pattern is the *Transition on Modes* pattern, in which the processes change their states based on a combination of their respective modes and the messages they receive on their inboxes. For example, in a distributed accounting system, a money receipt message would cause different ledgers to be modified, based on whether the controlling process was in "accounts receivable" or "accounts payable" mode.

Another pattern is the *Transition on Functions* pattern, in which the processes change their states based on a function of the information contained within the messages they receive on their inboxes. For example, in a distributed accounting system, an income transfer may require different actions based on how much money is being transferred, for tax shelter purposes.

6 Framework Implementation

Version 1.0 of our tools and models, released in February 1997, is classified in the "white box framework" level of the taxonomy given by the framework pattern language [18]. With the addition of more applications, services, visual builders, and language tools, we are developing a "black box framework." To guarantee widespread, unrestricted use, our framework has been developed using Sun's Java Developer's Kit (JDK) 1.0.2.

We are optimizing the framework for JDK 1.1 by taking advantage of the following newly standardized packages:

- *Remote Method Invocation* (RMI) for a proxy-based distributed object model.
- *Object Serialization* facilities for packing and unpacking objects and messages (both for communication and for persistent storage).
- *Java Database Connectivity* support for persistent storage of, and queries on, process, state, and interface data.
- *Interface Definition Language* (IDL) packages for interoperability with CORBA distributed objects.
- *Security* packages for communication encryption and process authentication.
- *Reflection* packages for innovative structuring of emergent personal networks and process behavior.

7 Related Work

Frameworks are reusable designs for software system processes, described by the combination of a set of objects and the way those objects can be used [18]. Our framework consists of some middleware APIs, a model for using them, and services and patterns that are helpful not only in inheriting from objects, but extending them as well. These features allow the reuse of both design and code, reducing the effort required to develop an application. In this sense, our framework is comparable to other metacomputing, component, and concurrency frameworks.

Metacomputing Frameworks. Our framework efforts are similar to recent metacomputing endeavors in that we use the Internet as a resource for concurrent computations. *Globus* provides the infrastructure to create networked virtual supercomputers for running applications [6]. Similarly, *NPAC at Syracuse* seeks to perform High Performance Computing and Communications (HPCC) activities using a Web-enabled concurrent virtual machine [7]. *Javelin* is a Java-based architecture for writing parallel programs, implemented over Internet hosts, clients, and brokers [2]. *Legion* is a C++-based architecture and object model for providing the illusion of a single virtual machine to users for wide-area parallel processing [10]. Although our framework could be used for metacomputing applications, we provide neither seamless parallelism, nor facilities for developing high-performance appplications. Rather, we provide mechanisms for programmers to develop distributed system components and personal networks quickly, and we plan to provide mechanisms for non-programmers to customize their components and their personal networks easily.

Component Frameworks. Many other framework systems also have the goal of creating distributed system components. The *ADAPTIVE Communication Environment* (ACE) provides an integrated framework of reusable C++ wrappers and components that perform common communications software tasks [19]; this framework is amenable to a design pattern group useful to many object-oriented communication systems [20]. *Hector* is a Python-based distributed object framework that provides a communications transparency layer enabling negotiation of communication protocol qualities, comprehensive support services for application objects, and a four-tiered architecture for interaction [1]. *Aglets* provide a Java-based framework for secure Internet agents that are mobile, moving state along with the program components themselves [13]. We differ from these efforts because our emphasis is on reasoning about global compositional distributed systems.

Concurrency Frameworks. We have considered several previous approaches to concurrent communicating processes in developing our framework. The *Communicating Sequential Processes* (CSP) model assumes each process is active for the entire duration of the computation [11]. Like Fortran M [5], we implement this model, adding such implementation artifacts as dealing with process setup and removal, and permitting prioritized waits to resolve resource contention. Unlike Fortran M, sessions provide a hybrid technique for running communicating distributed processes that are frozen when they are not performing any work, yet have persistent state that can be revived whenever a new session is initiated.

This persistence model is similar to mechanisms provided as recent ORB services [21]. However, the *CORBA* process model, implemented using the Basic Object Adaptor (BOA) of a given Object Request Broker (ORB), maintains that only the broker stay active for the entire duration of the computation [16]. Like *Client-Server*, *Remote Procedure Call*, and *Remote Method Invocation* systems, CORBA only spawns remote processes to perform isolated remote tasks. In our framework, the model supports interaction not just through a broker or server, but also directly between the ports of distributed processes in a peer-to-peer fashion.

8 Summary

In this paper, we have presented a framework for developing personal networks and their component processes, for using those processes in sessions to perform distributed tasks, and for reasoning about those processes and sessions. Our approach is novel in its simplicity, scalability, and flexibility; new system processes can be developed by inheriting from framework processes or by extending framework services.

In further research, we plan on using the framework to develop more substantial personal networks, including several task forces: research consortia that use instruments, computation engines, and visualization devices at different sites; oversight committees for conferences or journal publications; and working groups whose members hail from different organizations. In addition, we plan to investigate an array of services for use with the framework, including tools for active process mobility, distributed collaboration, termination detection, resource management, and session coordination.

References

[1] D. Arnold, A. Bond, M. Chilvers, and R. Taylor, 'Hector: Distributed Objects in Python', *Proceedings of the Fourth International Python Conference*, Livermore, California, June 1996.

[2] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu, 'Javelin: Internet-Based Parallel Computing Using Java', submitted to *Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1997.

[3] K. M. Chandy, A. Rifkin, P. A. G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman, 'A World-Wide Distributed Sytem Using Java and the Internet', *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, August 1996.

[4] K. M. Chandy and A. Rifkin, 'Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions', *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, Maui, Hawaii, January 1997.

[5] I. T. Foster and K. M. Chandy", 'Fortran M: A Language for Modular Parallel Programming'. *Journal of Parallel and Distributed Computing*, Volume 26, Number 1, Pages 24–35, April 1995.

[6] I. Foster and C. Kesselman, 'Globus: A Metacomputing Infrastructure Toolkit', *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing*, SIAM, Lyon, France, August 1996.

[7] G. Fox and W. Furmanski, 'Towards Web/Java based High Performance Distributed Computing – An Evolving Virtual Machine', *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, August 1996.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[9] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[10] A. S. Grimshaw, W. A. Wulf, and the Legion team, 'The Legion Vision of a Worldwide Virtual Computer', *Communications of the ACM*, Volume 40, Number 1, Pages 39–45, January 1997.

[11] C. A. R. Hoare, 'Communicating Sequential Processes', *Communications of the ACM*, Volume 21, Number 8, Pages 666–677, August 1978.

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

[13] D. B. Lange and M. Oshima, *Programming Mobile Agents in Java — With the Java Aglet API*, IBM Research, 1997.

[14] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1996.

[15] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete, *Atomic Transactions*, Morgan Kaufmann, 1994.

[16] Object Management Group, *The Common Object Request Broker: Architecture and Specification (CORBA)*, revision 2.0, 1995.

[17] L. L. Peterson and B. S. Davie, *Computer Networks: A Systems Approach*, Morgan Kaufmann, 1996.

[18] D. Roberts and R. Johnson, 'Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks', *Proceedings of Pattern Languages of Programs*, Allerton Park, Illinois, September 1996.

[19] D. C. Schmidt, 'ACE: an Object-Oriented Framework for Developing Distributed Applications', *Proceedings of the Sixth USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994.

[20] D. C. Schmidt, 'A Family of Design Patterns for Application Level Gateways', *Theory and Practice of Object Systems*, Wiley and Sons, Volume 2, Number 1, 1996.

[21] R. Sessions, *Object Persistence Beyond Object-Oriented Databases*, Prentice Hall, 1996.

[22] Workflow Management Coalition, International Organization for the Development and Promotion of Workflow Standards, *Workflow Glossary*, Workflow Management Coalition, Belgium, 1995.