A World-Wide Distributed System Using Java and the Internet

K. Mani Chandy, Boris Dimitrov, Huy Le, Jacob Mandelson, Matthew Richardson,
Adam Rifkin, Paolo A.G. Sivilotti, Wesley Tanaka, and Luke Weisman *
California Institute of Technology 256-80
Pasadena, California 91125
mani@cs.caltech.edu

March 8, 1996

Abstract

This paper describes the design of a distributed system built using Java that supports peer-to-peer communication among processes spread across a network. We identify the requirements of a software layer that supports distributed computing, and we propose a design that meets those requirements. Our primary concern is the identification, specification, and implementation of software components that can be composed in different ways to develop correct distributed applications. Though our implementation uses Java, the fundamental ideas apply to any object-oriented language that supports messaging and threads.

Keywords: Distributed systems, collaborative environments.

1 Introduction

Overview. Millions use the World-Wide Web for information exchange and for client-server applications. Widespread use has led to the development of a cottage industry for producing Web-based documentation; large numbers of people without formal education in computing are developing server applications on the Web. This paper describes a project to extend this infrastructure to a distributed system with *peer-to-peer process communication* across the Global Information Infrastructure (GII). The focus of our project is on identifying and specifying software components that can be composed to create distributed applications, implementing the software components as classes in an object-oriented framework, developing a compositional methodology for constructing correct distributed applications from the components, and implementing a library of applications that demonstrates the methodology.

Our project develops methods and tools for distributed programming applications layered on top of standard network technologies. Though our implementation uses Java, we could have used any other object-oriented language that supports threads and communication classes.

Correctness. Our project deals with providing software components and compositional methods that support the development of correct distributed applications. The methods employed by Web users for developing client-server applications are not the best methods for developing correct peer-to-peer distributed applications.

*This work is supported in part by the Center for Research on Parallel Computation under grant NSF CCR-912008, and by CISE Directorate grant CCR-9527130 for the integration of symbolic computing with frameworks of classes and problem-solving archetypes.

Furthermore, approaches for debugging sequential programs are inadequate for ensuring correctness in distributed applications. Our challenge is to deal with the difficult problems of distributed systems — problems such as deadlock, livelock, and sending unbounded numbers of messages — that are not issues in sequential programs.

Contrast with Traditional Distributed Systems. Certain kinds of distributed systems are inherently complex, deal with myriad functions, have strict performance requirements, and have disastrous consequences of failure; examples of these systems include applications in telemedicine, air traffic control, and military command. Such systems are developed in a painstaking manner, with each system design led by a single group of expert designers that has primary responsibility for the entire system. By contrast, many Web-based applications are relatively simple, are collaborative by nature, have limited functionality, are performance-limited by the GII, and may be developed by people who are not experts in concurrent computing; examples of these systems are the calendar application and collaborative environments described in section 2.1. Designers of such applications have little control over the networks, protocols, operating systems, and computers on which their applications execute.

Research issues for the two classes of distributed systems are somewhat different. The former class of distributed systems, the class on which human lives depend, is extremely important and has benefited from a great deal of research (c.f., the survey of critical distributed systems issues in [4]). This paper, however, deals with facilitating the development of the latter class of applications. The class of collaborative Web-based applications, though less critical than traditional distributed applications, has interesting engineering challenges nonetheless.

Dapplets and Sessions. We coin the phrase *dapplet* to distinguish a process used in a collaborative distributed application from processes used in traditional distributed systems. Dapplets are composed together to form distributed sessions. A *session* is a temporary network of dapplets that carries out a task such as arranging a meeting time for a group of people. Sessions need not be static; after initiation, they may grow and shrink as required by the dapplets. In the next section, we discuss dapplets and sessions, and propose requirements for them.

2 Requirements

We describe two simple example applications to motivate a Web-based distributed system, and present a requirements analysis for such a system and a design for meeting those requirements.

2.1 Two Example Applications

Example One: A Calendar Application. A consortium of institutions forms a research center, and the executive committee of the center has members from its component institutions. The director of the center wants to pick a date and place for a meeting of the executive committee. Several algorithms (c.f., [4]) can be used to solve this problem.

The traditional approach has the director (or someone on the staff) call each member of the committee repeatedly, and negotiate with each one in turn until an agreement is reached. The approach we propose (as illustrated in figure 1) is to employ secretary and calendar processes — programs running concurrently on each committee member's desktop computer — to suggest a set of candidate dates that can then be approved or rejected by the members.

Each member of the committee has a calendar process — a dapplet — responsible for managing that member's

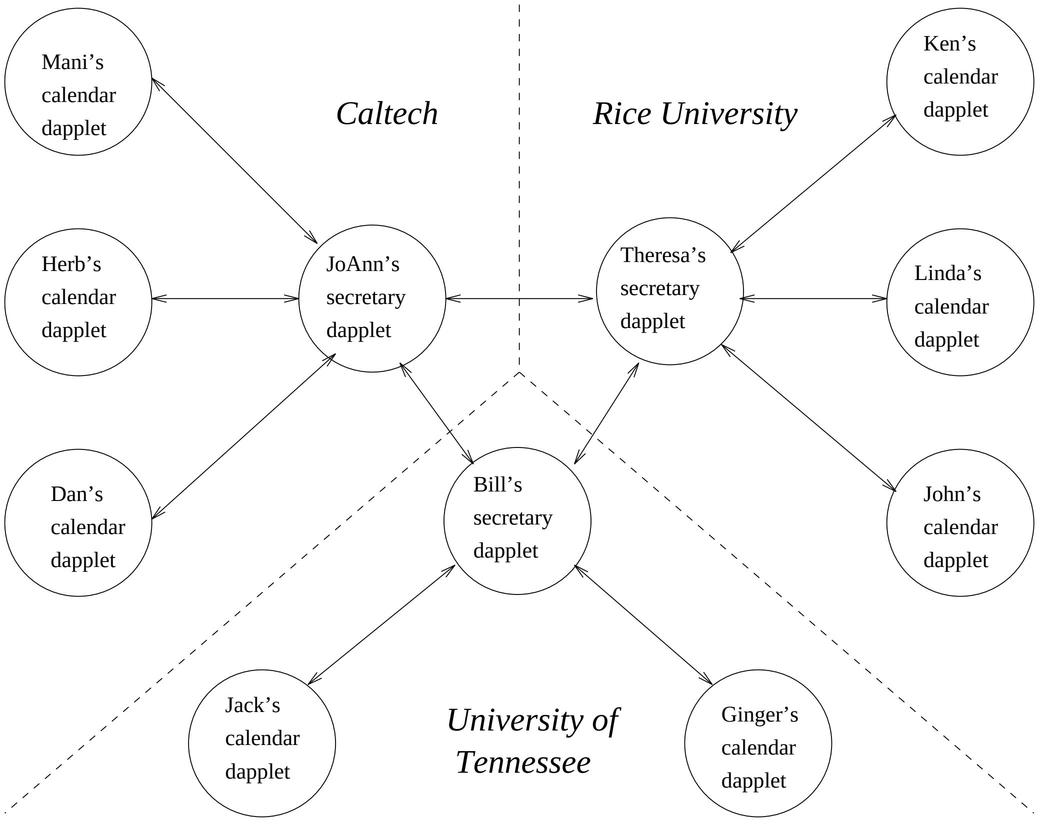

Figure 1: A session coordinating committee members at Caltech, Rice, and Tennessee. Note that each dapplet is running on a different computer, and the arrowed lines represent communication between distributed processes over the Internet.

calendar. There may in addition be secretary dapplets (as shown in figure 1), or possibly a coordinator dapplet. The dapplets are composed together into a temporary network of dapplets that we call a session. The task of the session is to arrange a common meeting time. When this task is achieved, the session terminates.

Example Two: Collaborative Distributed Design. A group of people, working at different sites, collaborate on the design of a system. Management of design documents requires that modifications to parts of the document are communicated to appropriate members of the design team. As the use of consultants and virtual corporations becomes more common, the need for "wiring up" different groups of people in different ways for different tasks becomes more important.

Each member of the design team has a dapplet responsible for managing that member's part of the design. The collection of dapplets forms a network — a session — that lasts as long as the design.

2.2 Characteristics of Applications

Temporary Duration. In many collaborative applications, a distributed session is set up for a period of time, after which the session is discarded. For instance, calendar dapplets of the executive committee are linked together into a dapplet-network session, and after the dapplets agree on a meeting date and time, the session is cancelled. Some distributed sessions may have longer duration. For instance, in our second example, the distributed session of participants in a system design lasts as long as the design.

Durations of distributed sessions in collaborative applications can vary widely. By contrast, traditional distributed systems such as command and control systems are semi-permanent. The challenge is to develop a software layer that supports distributed sessions with wide variations in duration.

Persistent State Across Multiple Temporary Sessions. In our first example, the state of an executive committee member's appointments calendar must persist; an appointments calendar that disappears when an appointment is made has no value.

Different parts of the state may be accessed and modified by different distributed sessions. For instance, a distributed session to set up an executive committee meeting may have access to Mondays and Fridays on one user's calendar, but not to other days, and a distributed session to inform collaborators about the status of a document may have access to document information but not to the calendar.

The state of a process may be accessed and modified by multiple concurrent sessions. Each session (e.g., a calendar session or a document management session) only has access to portions of the state relevant to that session. The specification of a session must be independent of other sessions with which it may be executing concurrently. Two sessions must not be allowed to proceed concurrently if one modifies variables accessed by the other.

The challenge is to provide the distributed infrastructure that sets up sessions that modify the persistent states of their participants, allows a member to participate in concurrent sessions, and ensures that sessions that interfere with each other are not scheduled concurrently.

Composition of Services. A traditional distributed system [11] is architected in a series of well-defined layers, with each layer providing services to the layer above it and using services of the layer below. For instance, a distributed database application employs services — e.g., checkpointing, deadlock detection, and transaction abortion — of the distributed operating system on which it runs.

A session also needs operating system services. The model of application development for sessions and dapplets is, however, very different from that in traditional systems. We do not expect each dapplet developer to also develop all the operating systems services — e.g., checkpointing, termination detection, and multiway synchronization — that an application needs. Our challenge is to facilitate the development of a library of operating systems services (which we could call *servlets*) that dapplet developers could use in their dapplets, as needed.

Coping with a Varied Network Environment. Communication delays can vary widely. One process in a calendar application may be in Australia while two other processes are in the same building in Pasadena. Our challenge is to design the system to cope with these delays; in addition, the system must also cope with faults in the network such as undelivered messages.

Patterns of Collaboration. In distributed applications, it is more difficult to verify the correctness of the concurrent and distributed aspects than it is to verify the sequential programming aspects. The difficult parts of a distributed system design include the correct implementations of process creation, communication, and synchronization. However, we can ease the programmer's burden of writing correct distributed applications, if modifying one distributed application to obtain another one with the same patterns of communication and synchronization can be done by modifying only the sequential parts of the application while leaving the concurrent and distributed parts unchanged. Our challenge is to identify these patterns, develop class libraries that encapsulate these patterns, and construct a library of distributed applications that demonstrate how common collaboration patterns can be tailored to solve a specific problem by modifying the sequential components.

3 Distributed System Design

3.1 Intended System Use

Consider the example of a center director setting up an executive committee meeting with members from different sites. Prior to the session, each committee member has installed a calendar dapplet. A calendar dapplet is a process: it operates in a single address space, it communicates with files by standard I/O operations, and it communicates with other processes through ports. Associated with each dapplet is an Internet address (i.e., IP address and port id); the port id may change, but that is not our concern at this stage in the discussion.

A session is an instance of an application, implemented as a network of dapplets. A session consists of many different types of dapplets. For instance, a calendar application may have calendar user processes and secretary processes. Programs corresponding to each process type are installed on the appropriate machines; for the session in figure 2, the calendar user dapplets and secretary dapplets are processes running on their respective users' desktop computers.

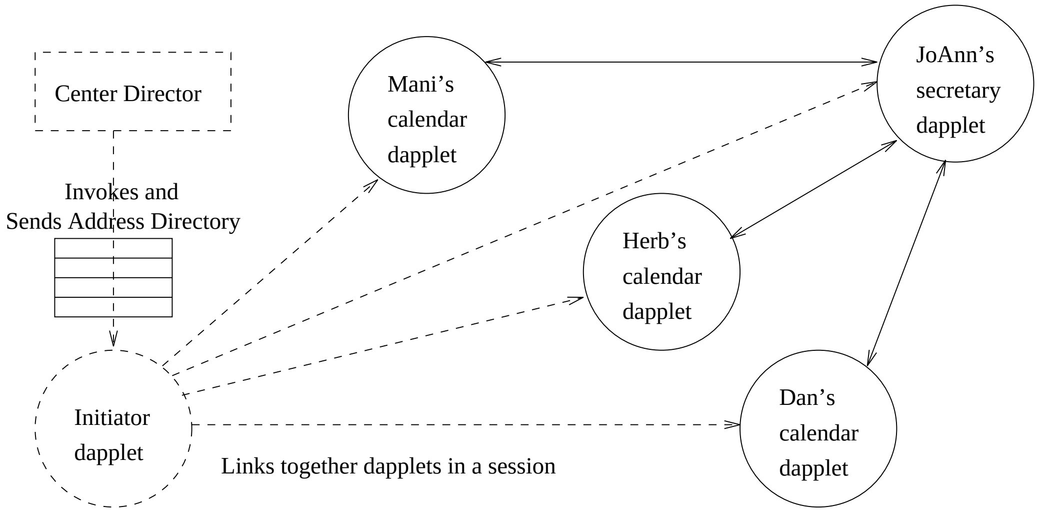

Figure 2: An initiator uses the invoker's address directory to set up a session between existing dapplets.

Associated with each session is an initial process — an *initiator dapplet* — that is responsible for linking dapplets together. In our example, the center director invokes an initiator dapplet, and passes it a directory of addresses (e.g., Internet IP addresses and ports) of component dapplets that are to be linked together into a session, as illustrated in figure 2. We do not address how this directory is maintained in this paper.

Dapplet connections are achieved using the address directory. The initiator dapplet sends a request to the component dapplets; this request asks the components to link themselves up to form a session. For example, in our calendar session, each calendar user dapplet may be linked to a common coordinating secretary dapplet, as is done in figure 2. As another example, in a distributed card game session, a player dapplet may be linked to its predecessor and successor player dapplets (which correspond to the players to its left and right, respectively).

A dapplet, on receiving a request to participate in a session, may accept the request and link itself up, or it may reject the request (because the requesting dapplet was not on its access control list, or because it is already participating in a session and another concurrent session would cause interference). We postpone consideration of which actions the initiator could take if a session cannot be established. When a session terminates, component dapplets unlink themselves from each other.

3.2 Overall Distributed System Design

Our distributed system implementation is written in the Java language [6], and uses Java socket classes [7] and thread primitives [1]. The initial implementation uses UDP [9, 10], and it includes a layer to ensure that messages are delivered in the order they were sent.

We describe the overall design, and highlight the software components we believe are useful for developing distributed applications. Our goal is to design a simple layer to support correct distributed application development; we employ Java features and Java classes to achieve this end.

Messages. Objects that are sent from one process to another are subclasses of a *message* class. An object that is sent by a process is converted into a string, sent across the network, and then reconstructed back into its original type by the receiving process. Java methods are used to convert an object to a string and to create an instance of the sending object at the receiver.

Inboxes, Outboxes, and Channels. Each process has a set of *inboxes* and a set of *outboxes*. Inboxes and outboxes are message queues. A process can append a message to the tail of one of its outboxes, and it can remove the message at the head of one of its inboxes. The methods that can be invoked on inbox and outbox objects are described later. Each inbox has a global address: the address of its dapplet (i.e., its IP address and port) and a local reference within the dapplet process.

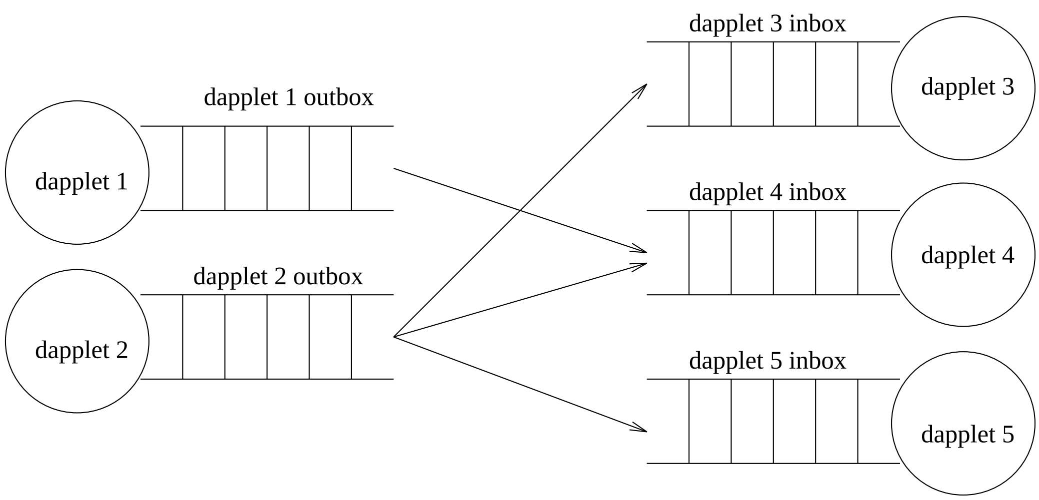

Figure 3: An example of dapplet inbox and outbox connections: dapplet 1's outbox is bound to dapplet 4's inbox; dapplet 2's outbox is bound to the inboxes of dapplets 3, 4, and 5.

Associated with each outbox is a set of inboxes to which the outbox is bound; there is a *message channel* from an outbox to each inbox to which it is bound; an example of a set of bound dapplet inboxes and outboxes is given in figure 3. Each message channel is directed from exactly one outbox to exactly one inbox. Messages sent along a channel are delivered in the order sent. Message delays in channels are arbitrary.

As shown in figure 3, an outbox can be bound to an arbitrary number of inboxes. Likewise, an inbox can be bound to an arbitrary number of outboxes. Therefore, there are an arbitrary number of outgoing channels from an outbox, and there are an arbitrary number of incoming channels to an inbox.

The distributed computing layer removes the message at the head of a nonempty outbox and sends a copy of the message along all channels connected to that outbox. The network layer delivers a message in a channel to the destination inbox of the channel. The delay incurred by a message on a channel is arbitrary; the delay is independent of the delay experienced by other messages on that channel, and it is independent of the delay on other channels. Also, if a message is not delivered within a specified time, an exception is raised.

Methods Invoked on Outboxes. An outbox has a data member `inboxes`, which is a list of addresses of inboxes to which the outbox is bound. The application-layer methods that can be invoked on outboxes are:

1. `add(ipa)` where `ipa` is the global address of an inbox; this method appends the specified inbox to the list `inboxes` if it is not already on the list. There is a directed FIFO channel from each outbox to each inbox to which it is bound.
2. `delete(ipa)` removes the specified global address from the list `inboxes` if it is in the list, and otherwise throws an exception.
3. `send(msg)` where `msg` belongs to a subclass of message; this method sends a copy of the object `msg` along each output channel connected to the outbox. If this message is not delivered within a specified time, an exception is raised.
4. `destination()` returns `inboxes`.

Methods Invoked on Inboxes. The application-layer methods that can be invoked on inboxes are:

1. `isEmpty()` returns `true` if the inbox is empty.
2. `awaitNonEmpty()` suspends execution until the inbox is nonempty.
3. `receive()` suspends execution until the inbox is nonempty and then returns the object at the head of the inbox, deleting the object from the inbox.

Strings as Names for Inboxes. As a convenience, we also allow each inbox to be addressed by a pair: its unique dapplet address (IP address and port) and a string in place of its local id. For instance, a professor dapplet may have inboxes called "students" and "grades" in addition to inboxes to which no strings are attached. An outbox of a dapplet can be bound to the "student" inbox of a professor dapplet. The `add` and `delete` methods of a dapplet are polymorphic: an inbox can be either specified by a global address (dapplet address and local reference) or by a dapplet address and string.

Communication Layer Features. Our simple communication layer, when used with objects and threads, can provide features present in more complex systems.

Some languages, such as CC++ [2], have a two-level hierarchy of address spaces: a global address space and a collection of local address spaces. So, pointers are of two kinds: global and local. A *global pointer* in one local address space can point to an object in any local address space. By contrast, a *local pointer* in a local address space can point only to objects in that local address space. Remote procedure calls (*RPCs*) on an object in a different local address space can be executed only if the invoker has a global reference to that object.

By contrast, in our Java implementation, all references are local, with the exception that dapplets and inboxes have global addresses. An outbox in one dapplet can bind to inboxes in other dapplets. Addresses of inboxes and dapplets can be communicated between dapplets.

Global pointers and RPCs are implemented in our Java system in a straightforward way: Associate an inbox `b` with an object `p`. Messages in `b` are directions to invoke appropriate methods on `p`. Associate a thread with `b` and `p`; the thread receives a message from `b`, and then invokes the method specified in the message on `p`. Thus the address of the inbox serves as a global pointer to an object associated with the inbox, and messages serve the role of asynchronous RPCs. Synchronous RPCs are implemented as pairwise asynchronous RPCs.

4 Software Components

We consider the problem of composing services with dapplets. The challenge is to make these services generic so that they can be used for very different kinds of applications, and make the services powerful enough to simplify the design of dapplets.

We focus our discussion here on inter-dapplet services. Methods for coordination within a dapplet use standard Java classes [5]. The questions we address are: How can objects associated with a service be bound into a dapplet in a straightforward way, and, what sorts of services are helpful for dapplet designers?

There are complementary ways of providing services to dapplets. We can provide a collection of service objects that a designer can include in a dapplet. In addition, we can have a resource manager process, executing on each machine, that provides a rich collection of services to dapplets executing on that machine. Our focus in this paper is on the former approach; we give a few examples of service objects and show how these services can be used within a dapplet.

4.1 Tokens and Capabilities

Distributed operating systems manage indivisible resources shared by processes [11]; we would like to provide service objects with this functionality, which a dapplet designer can incorporate as needed. A problem is that generic service objects do not have information about the specific resources used in a given application.

A solution is to treat indivisible resources in a generic way. The generic service deals with managing indivisible resources, sharing them among dapplets in a way that avoids deadlock (if dapplets release all resources before next requesting resources), and detecting deadlock if it does occur (if a dapplet holds on to some resources and then requests more). The designer of a dapplet can separate these service functions from other concerns, and using a library of common service functions can simplify dapplet design.

We treat each resource as a *token*. Tokens are objects that are neither created nor destroyed; a fixed number of them are communicated and shared among the processes of a system. Tokens have colors; tokens of one color cannot be transmuted into tokens of another color. A token represents an indivisible resource, and a token color is a resource type. A file, for instance, is represented by a token and each file-token has a unique color.

A network of token-manager objects manages tokens shared by all the dapplets in a session. A token is either held by a dapplet or by the network of token managers. A token manager associated with a dapplet has a data member, `holdsTokens`, which is the number of tokens of each color that the dapplet holds.

A process can carry out the following operations on its token manager.

1. `request(tokenList)` suspends until the requested tokens (i.e., a specified number for each color) is available, and then these tokens are removed from the token manager collection and given to the dapplet (i.e., these tokens are added to `holdsTokens`). If the token managers detect a deadlock, an exception is raised. A specific positive number of tokens of a given color can be requested or the request can ask for *all* tokens of a given color.

2. `release(tokenList)` releases the specified tokens from the dapplet and returns them to the token managers; therefore, the specified tokens are decremented from `holdsTokens` and returned to the token managers. If the tokens specified in `tokenList` are not in `holdsTokens`, an exception is raised.

3. `totalTokens()` returns an array of the total number of tokens of all colors in the system.

The dapplet that constructs the network of token managers ensures that the initial number of tokens is set appropriately. Tokens are defined by the invariant that the total number of tokens of each color in the system remains unchanged.

Tokens can be used in many ways. For example, suppose we want at most one process to modify an object at any point in the computation. We associate a single token with that object, and only the process holding the token can modify the object.

As another example, tokens can be used to implement a simple read/write control protocol that allows multiple concurrent reads of an object but at most one concurrent write (and no reads concurrent with a write) of the object. The object is associated with a token color. A dapplet writes the object only if it has all tokens associated with the object, and a dapplet reads the object only if it has at least one token associated with the object.

4.2 Clocks

Access to a global clock simplifies the design of many distributed algorithms. For instance, a global state can be easily checkpointed: all processes checkpoint their local states at some predetermined time T, and the states of the channels are the sequences of messages sent on the channels before T and received after T.

Another use of global clocks is in distributed conflict resolution. Each request for a set of resources is timestamped with the time at which the request is made. Conflicts between two or more requests for a common indivisible resource are resolved in favor of the request with the earlier timestamp. Ties are broken in favor of the process with the lower id. If dapplets release all resources before requesting resources, and release all resouces within finite time, then all requests will be satisfied.

The problem is that dapplets do not share a global clock. Though local clocks are quite accurate they are not perfectly synchronized. We can, however, use unsynchronized clocks for checkpointing provided they satisfy the global snapshot criterion [3]. The global snapshot criterion is satisfied provided every message that is sent when the senders clock is T is received when the receiver's clock exceeds T. A simple algorithm [8] to establish this criterion is: every message is timestamped with the sender's clock; upon receiving a message, if the receiver's clock value does not exceed the timestamp of the message, then the receiver's clock is set to a value greater than the timestamp.

Our message-passing layer is designed to provide local clocks that satisfy the global snapshot criterion. Our local clocks can be used for checkpointing and conflict resolution just as though they were global clocks. Dapplet designers can separate the generic concerns of clock synchronization from other concerns specific to their application.

4.3 Synchronization Constructs

Java provides constructs for synchronizing threads within a dapplet by using something like a monitor [7]. We have implemented and verified other kinds of synchronization constructs — barriers, single-assignment variables, channels, and semaphores — for threads within a dapplet [5]. We are extending these designs to allow synchronizations between threads in different dapplets in different address spaces.

5 Summary

This paper identifies a class of distributed systems that is different from the traditional variety, and gives an analysis of the requirements of such systems. We propose a design to meet the requirements. The design supports distributed sessions of processes (dapplets) that can be composed by linking input and output ports (inboxes and outboxes). The design was selected because it supports modularity — each dapplet has a simple interface defined by its ports — and because its implementation in Java using Java socket classes is straightforward. The design facilitates the creation and maintenance of sessions — temporary networks of dapplets that carry out specific tasks. Our work focuses on designing, implementing, and verifying services that can be composed within dapplets; we described some services that simplify the design of dapplets.

References

[1] A. D. Birrell. An introduction to programming with threads. Technical Report Report 35, Digital Systems Research Center, 1989.

[2] K. M. Chandy and C. Kesselman. Cc++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: Determining the global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, February 1985.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[5] K. M. Chandy and P. A. G. Sivilotti. Toward high confidence distributed programming with java: Reliable thread libraries. In *International Conference on Software Engineering*, 1996.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Developers Press, Sunsoft Java Series, 1996.

[7] J. Gosling, F. Yellin, and the Java Team. *The Java Application Programming Interface*. Addison-Wesley Developers Press, Sunsoft Java Series, 1996.
See also `http://java.sun.com/JDK-beta2/api/java.net.Socket.html`

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[9] J. Postel. *User Datagram Protocol*. RFC 768, 3 pages, August 1990.

[10] W. R. Stevens. *Unix Network Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[11] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1995.