Systematic Composition of Objects in Distributed Internet Applications: Processes and Sessions

K. Mani Chandy and Adam Rifkin *
Computer Science 256-80
California Institute of Technology
Pasadena, California 91125
{*adam,mani*}*@cs.caltech.edu*

June 9, 1996

Abstract

We consider a system with the infrastructure for the creation and interconnection of large numbers of distributed persistent objects. This system is exemplified by the Internet: potentially, every appliance and document on the Internet has both persistent state and the ability to interact with large numbers of other appliances and documents on the Internet. This paper elucidates the characteristics of such a system, and proposes the compositional requirements of its corresponding infrastructure. We explore the problems of specifying, composing, reasoning about, and implementing applications in such a system. A specific concern of our research is developing the infrastructure to support structuring distributed applications by using sequential, choice, and parallel composition, in the anarchic environment where application compositions may be unforeseeable, and interactions may be unknown prior to actually occurring. The structuring concepts discussed are relevant to a wide range of distributed applications; our implementation is illustrated with collaborative Java processes interacting over the Internet, but the methodology provided can be applied independent of specific platforms.

Key words: Distributed Systems, Composition, Reasoning, Java, Internet.

*This research is supported in part by NSF grants CCR-912008 and CCR-9527130. This work constitutes part of the Caltech Infospheres Project; more information is available on the Web at `http://www.infospheres.caltech.edu/`.

1 Introduction

This paper deals with the problems of specifying, composing, reasoning about, and implementing collaborative Internet-based applications. As the use of the Internet for collaboration continues to grow, the class of problems associated with such applications becomes increasingly important. The issues encountered in designing such applications are somewhat different from those encountered in traditional structured distributed systems. This section motivates work on these problems.

1.1 The Problem Domain

Traditional distributed systems (e.g., air-traffic control), are constructed with reliability in mind; for such systems, the consequences of failure would be disastrous. To ensure reliability control, such a system is developed and maintained with the overall responsibility designated to a single agency (in the case of our example, the Federal Aviation Administration). By contrast, a collaborative application on the Internet may be composed of many program units developed by different groups of people. For such distributed systems, no single agency assumes overall responsibility for reliability control. For convenience, we refer to the former kind of concurrent system as *structured*, as opposed to the latter kind, which we call *anarchic*.

A Simple Example. Figure 1 provides a small application that illustrates the problems of specifying, composing, reasoning about, and implementing collaborative Internet-based applications. Suppose an interest group on collaborative applications is considering holding a "Birds of a Feather" (*BOF*) meeting at the HICSS conference. Each site-appointed secretary polls the group members to determine (a) whether they will be attending HICSS, and (b) if they are attending, the evenings during which they can attend a BOF meeting. Then, the secretaries coordinate with each other, generating a few potential meeting times, and each secretary checks with its respective group members about selecting one. This procedure may have to be repeated until the group converges on a date, or until they decide that no date is acceptable to a quorum of the members.

Presently, the procedure of choosing a BOF meeting time is usually carried out by email. An alternative approach is to carry out this procedure automatically using a distributed program (that might also verify with the group members as a final step). Our work is concerned with the theories and tools that facilitate developing such a distributed program. Next, we present a few examples of structured concurrent systems, and identify the issues that make anarchic systems different.

Structured and Anarchic Systems. Four very different applications that can be developed using the theories and tools of structured concurrent systems are: an air-traffic control system, a parallel simulation of fluid flow, a tool that enables computer-aided design of VLSI chips, and a database server with a collection of clients that perform queries and updates. Some of the issues that are different when developing anarchic applications (such as an automatic BOF meeting time scheduler) are:

1. In structured systems, the design proceeds from a specification, and there is a single entity that is ultimately responsible for the design and implementation of the system.

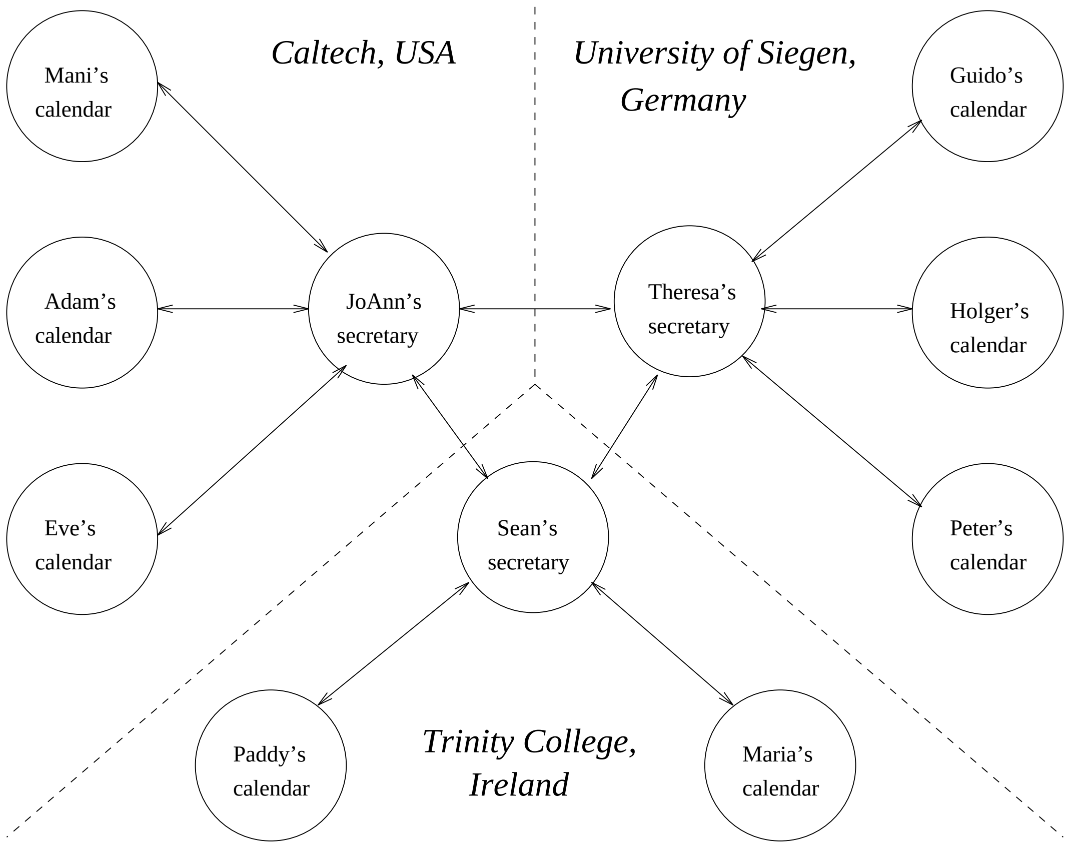

Figure 1: People at three different universities want to decide on a time to meet. Typically, this decision is achieved by appointing one or more secretaries to coordinate polling and meeting time selection.

By contrast, all of the processes on the Internet that modify calendars might not be drawn from the same specification. In the BOF example in Figure 1, there might not be a single agency responsible for designing and implementing the calendar processes of all the BOF members in the Internet.

2. The interactions between the components in the structured examples are specified as part of the design. For example, in the simulation of fluid flow, the designers use the specification to determine exactly what the components of the simulation are, how these components interact, and where the components are located.

   By contrast, a collaborative distributed application developer might not know which processes are going to interact before the interaction itself begins. Specifically, by providing a publicly accessible program interface available on the Internet, the developer allows interactions to occur between the local program and any other process on the Internet cognizant of that interface. A user of the program may have to figure out where a component (e.g., the calendar scheduling process of the BOF secretary) is located. Also, components can allow different checkable access privileges to various other components.

3. In the structured examples, an application program can be partitioned into components in systematic ways. The computer science community has discovered methodical ways of dividing a task into components, and composing those components using sequential, choice, and parallel composition (c.f., [CM88, CT92]).

   In the anarchic case, the components are given, and the task is to compose them to achieve some end. The application developer needs to determine whether components have compatible interfaces, and whether the components can be composed in a meaningful way. If they cannot be composed, the developer must address whether they can be refined in a straightforward way, so that composition is feasible.

A specific concern of our research is developing the infrastructure that supports structuring distributed applications by composing components using sequential, choice, and parallel composition, in the anarchic environment of unforeseen applications and unpredicted interactions.

1.2 Contributions of This Work

Structuring Collaborative Applications. We suggest program component units that can be composed in systematic ways to create structured distributed applications. We propose two kinds of compositional units, *processes* and *sessions*, and demonstrate properties of these units in the context of other work done on the theory of composition.

Processes can be composed in parallel, and we reason about processes using theories of parallel composition in temporal logic [CM88, Lam94]. Sessions are collections of processes composed in parallel [AL93, Cha94]. A session is specified in terms of the precondition and postcondition predicates [Hoa69] of its component processes. Sessions can be composed using sequential and choice composition, and we reason about sessions using theory from the field of sequential programming [DS90]. Distributed applications can be structured by nesting processes and sessions, and our software infrastructure [CRS$^+$96] supports such capability.

Composing Distributed Objects on the Internet. We model every program, appliance, and document connected to the Internet with a state that is persistent for the lifetime of its corresponding entity, as shown in Figure 2. In this model, as in any state-based approach, the execution of a system is regarded as a sequence of states, where a state is an assignment of values to a set of variables.

Given this model, we propose an infrastructure that supports the systematic modification of the states of arbitrary collections of entities. Specific questions that we explore include the following. How are processes and sessions specified? What is the interface between processes? How does a programmer deal with different processes having different capabilities with respect to other processes? What is parallel composition, i.e., how can processes be composed together? What are the rules that determine that processes can be composed, and what can be done if these rules are not satisfied? How can process types and specific instances of processes be found on the Internet? How are sessions composed, and how can sessions be nested within processes?

Programming Model. Our programming model is different from the traditional model used in distributed systems: each appliance and document on the Internet has a corresponding state,

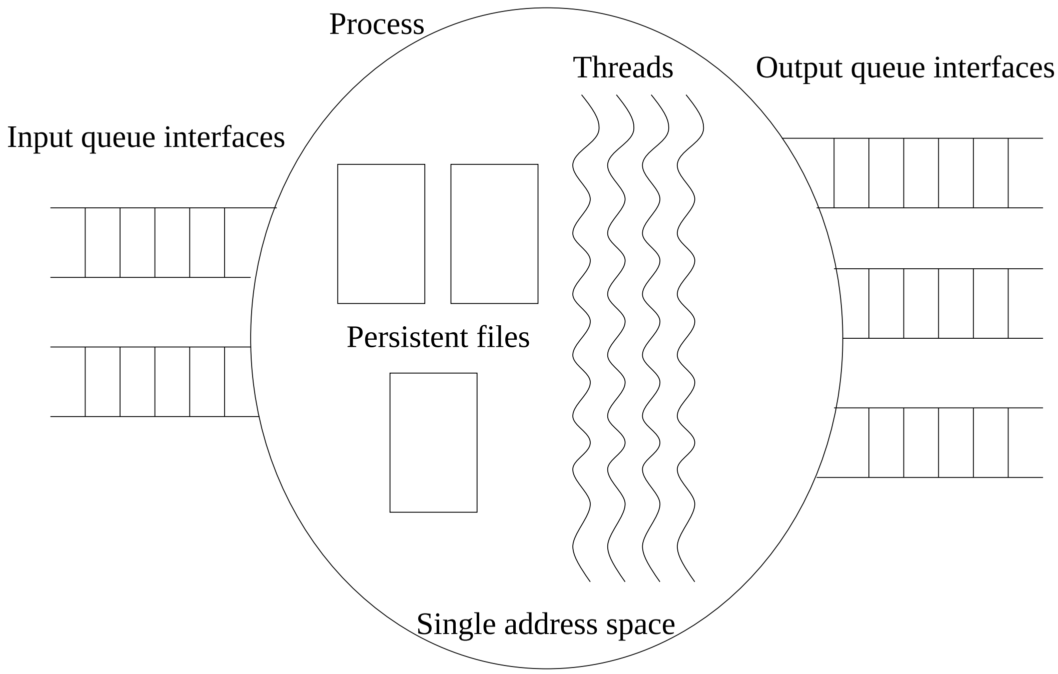

Figure 2: State is encapsulated in the persistent files of an entity's corresponding process. A process is a persistent object that may be multithreaded. Interfaces to other processes provide a mechanism for reliably transferring requests to modify local state.

and access to this state is handled by a controlling process. As a result, our overall distributed system may have millions of persistent objects and hundreds of thousands of concurrent sessions. One of the issues addressed in this paper is the provision of a programming model that has large numbers of processes while using limited resources.

One way in which collaboration among many processes can occur is by using the client-server paradigm: all processes are clients of a server process that is responsible for coordination. An alternative manner of collaboration is to have peer-to-peer interaction among processes, for which all processes are responsible for coordination. We conjecture that our programming model could handle both client-server and peer-to-peer interactions, though the focus in this paper is on peer-to-peer communication.

Implemented Infrastructure. Our Caltech group is designing and implementing an infrastructure [CRS$^+$96] based on the models and theories of structured composition discussed in this paper. The infrastructure allows application developers to design and implement collaborative processes and sessions over the Internet. Our implementation uses standard platforms that are widely available: Java [GJS96], TCP/IP [Ste94], and the World Wide Web [BLCGP92]. The focus of our research, however, is on basic ideas about composition applicable to any collaborative distributed system. As discussed in Section 5, CORBA [Obj95] can be employed to obtain a more elegant implementation, but our current system does not use this technology.

2 The Structure of Collaborative Applications

This section elaborates on the basic objects for composition introduced in Section 1.2: processes and sessions. Specification and composition mechanisms will be discussed in Section 3.

2.1 Processes and Sessions

Encapsulating State within Processes. In our underlying model, each document and appliance has a *state* that consists of a set of value assignments for that given entity's variables. In a reliable distributed system, the states of components should be modified only in systematic ways. For example, only authenticated processes should be permitted to modify the state of a given process (e.g., an appointments calendar). In addition, some processes may have privileges that other processes do not enjoy. For example, processes corresponding to the chair of a meeting may have privileges that processes of ordinary members do not possess. Furthermore, a reliable distributed system will provide safety mechanisms (e.g., a guarantee that disallows two appointments for the same person from being scheduled for exactly the same time).

To enable reliable application development, we encapsulate the state of an entity within a process that manages that entity (i.e., document or appliance), as illustrated in Figure 2. The state can be changed only by servicing requests received from other processes. From an implementation standpoint, each process is a multithreaded persistent Java object that can communicate with other processes using UDP [Pos80].

Requests to Modify State. A process cannot modify the state of another process directly; however, a process P can *request* a process Q, that Q modify its state in a manner prescribed by P, as illustrated in Figure 3. The kinds of requests that P can make of Q depends on the relationship between P and Q; for instance, if P is Q's boss, then P can make requests that Q's subordinates cannot make. The process structure facilitates communication of requests, and it also supports verification that requestors have appropriate capabilities.

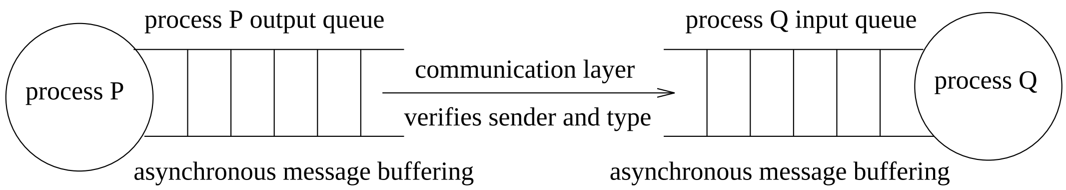

Figure 3: To modify the state of remote process Q, process P sends requests asynchronously.

Asynchronous Communication. A group of cooperating processes (i.e., a *session*) may be distributed on the Internet, anywhere in the world; in one session, all the processes might be in the same room, and in another session the processes might be on different continents. In some sessions, processes might have to interact with people during the session; for instance, a calendar process might need to get the acquiescence of the owner of the calendar before appointments are set. The time taken by a person to react to a signal from a process can vary significantly. Therefore, the delay between sending a message and the eventual response to the message can vary a great deal. For this reason, asynchronous buffered message-passing mechanisms are used, as illustrated in Figure 3.

Therefore, the underlying communication mechanism is not a synchronized remote procedure call (*RPC*) [BN84]; a process P cannot modify Q's state by executing an RPC on Q. Rather, a process P can send a message to Q requesting that Q execute an asynchronous (i.e., one-way) RPC, and this message is placed in one of process Q's incoming message queues. Process Q determines how its incoming queues are managed; for instance, it may give priority to one queue over another.

Our model provides a dynamic mechanism that allows processes to create and destroy new input and output queues during sessions, as illustrated in Figure 4.

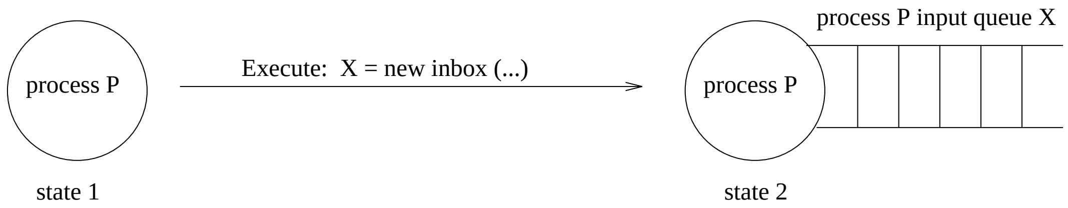

Figure 4: Process P creates a new input queue X.

Process Interfaces. A set of incoming message queues and a set of outgoing message queues is associated with each process. A *message queue* of a process P is a local object of P; message queues can be created or eliminated just like any other object.

As illustrated in Figure 5, an output message queue can be bound to an arbitrary number of input queues. A message at the head of an output queue is sent to every input queue to which it is bound, after which the message is deleted from the output queue; each input queue gets an identical copy of the message. Assume for the time being that all messages are delivered, even though the actual protocol (discussed briefly later) allows for dropped messages and employs timeouts.

Also as illustrated in Figure 5, an input queue can be bound to an arbitrary number of output queues. Messages from an output queue to an input queue are delivered in the order (i.e., first-in first-out or *FIFO*) that they are sent along the corresponding channel. The sequence of messages delivered to an input queue is a fair merge of the sequences of messages sent to the input queue from all the output queues to which it is bound.

Messages queues are *typed*; the type of a message queue specifies exactly which types of messages can be placed in the queue. An output queue is bound to an input queue only if any message type that can appear in the output queue can also appear in the input queue; we discuss more about binding later.

Process Capabilities. A process may have many input queues. Each input queue restricts the types of messages that can be placed in the queue. A set of processes is associated with each

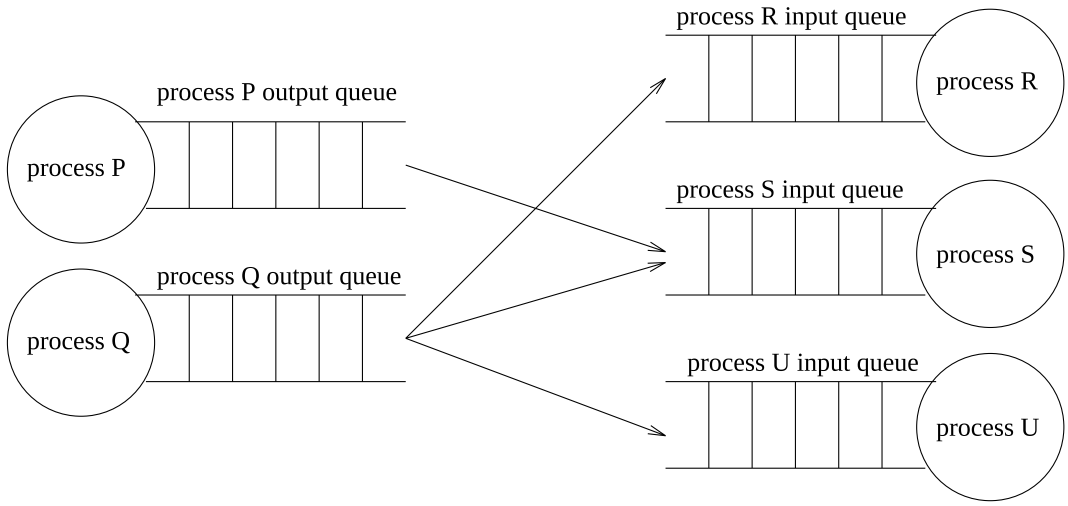

Figure 5: An example of process input and output queue connections: process P's output queue is bound to process S's input queue; process Q's output queue is bound to broadcast to the input queues of processes R, S, and T. Messages are fairly merged on process S's input queue. Our message-passing mechanism ensures FIFO delivery of messages on any given channel.

input queue, only the output queues of these processes can be bound to the input queue. In our implementation, this condition is ensured by the binding mechanism provided by our infrastructure [CRS+96]. Thus, the infrastructure facilitates control of messages that can be delivered to the input queue of a process. For instance, an input queue may restrict the binding to it to only allow "manager" processes.

This set of processes is specified either as an enumerated list or by attributes. Our current design allows the specification to be either a list or "any," but there are more sophisticated schemes that fit our overall plan; for instance, an input queue of type *colleague* of a person's calendar process can be restricted to be bound only to output queues of calendar processes of people in that person's work group.

A message sent to an input queue from an invalid output queue is not delivered; in our implementation, an exception is thrown in the sending process. Furthermore, only messages sent by processes in a specified set can be placed in the input queue. Our present design does not support security; for instance, it does not prevent a rogue process from pretending to be another process.

Parallel Composition of Processes. One of the methods that can be invoked on an output queue *binds* the output queue to a set of input queues. Binding an output queue to an input queue sets up a FIFO channel from the output queue to the input queue. *Parallel composition* among a set of processes can be achieved by binding the input and output queues appropriately; we discuss parallel composition of processes in Section 3.1. Our model provides a dynamic mechanism that allows the bindings of input and output queues to change during sessions, as shown in Figure 6.

Each input queue has a unique global address. This address is an IP address, socket number, and a local address for the input queue on its host processor. A process can bind one of its output

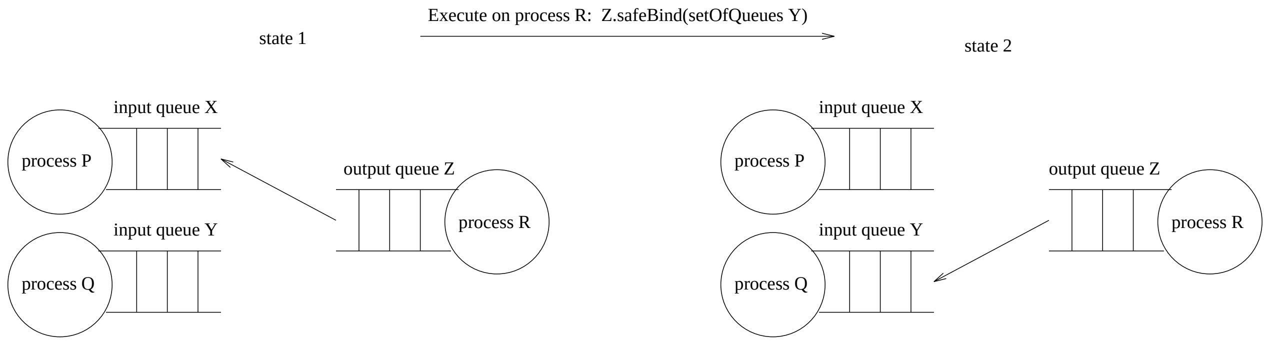

Figure 6: In changing the binding of an output queue to a different set of input queues (or, in this case, to a different single input queue), a process can perform a quickBind or a safeBind. Note that the process that has the output queue changes the binding, using an asynchronous request. The binding mechanism ensures reliable reconnections when a safeBind is called.

queues to an input queue of another process (or to one of its own input queues). The design has two kinds of bind methods: a *quickBind* and a *safeBind*. The quick version does not check the type and access control of the input queue to which it binds; if binding is invalid, an exception is thrown in the sending process when the first invalid message is sent along the channel. The safe version completes the binding only after checking that binding is valid in terms of type and access control, and an exception is thrown if the binding is invalid.

Every message includes in its header the identity of the process that sent the message and the message type. The communications layer delivers the message to an input queue only if the type and access control are valid. Each message is checked at the point of delivery to the input queue, because the access control list for an input queue can be changed at any point in the computation.

Implementation of Processes. In our implementation, a process is a Java program that has a collection of files (persistent storage) and which interacts with other processes by operations on its message queues. Since the program accesses files, it is implemented as a Java application program and not as an applet. The input and output message queues of a process are local objects of the process.

A method on an output queue (a) changes the set of input queues to which the output queue is bound either by binding another input queue using quick or safe binding or by deleting an input queue from the set, or (b) appends a message to the rear of the queue. A method on an input queue (a) changes the access control list for the queue by adding or removing processes from the list, (b) waits until the queue is nonempty and then returns the message at the head of the queue (and deletes this message from the queue), or (c) returns a value indicating whether the queue is empty or nonempty.

A process can do anything a Java program can do; for instance, it can have one thread for each input queue, where each thread waits for a message in its input queue. We do not restrict how a process handles messages or files. We are developing systematic ways for manipulating threads and messages [SC96], analyzing application performance [Rif96], and reasoning about parallel programs [Tho96, Mas97], but these methods are not discussed in this paper.

Process Persistence. If each appliance and document attached to the Internet is encapsulated within a process, a computer may have to support hundreds or thousands of persistent processes. Efficiency requirements limit the number of concurrently executing processes on a computer. Our scheduling layer limits the number of concurrently executing processes to those that are active — i.e., participating in sessions.

A request to a process to participate in a session is sent to the home address of the process where the request is trapped by a scheduler. If the process is already executing, the scheduler passes the message on to the process. If the process is not executing, the scheduler causes the process to execute (forks the process), and then passes it the message.

Process Mobility. Mobile processes are dealt with in the following way. Each process has an unchanging "home" address. This home address can be found by using search engines on the Web, for instance. The home address includes the unchanging address of an input queue to which requests to participate in sessions are sent. Thus, in phase 1 of session initiation, the initiator sends request messages to permanent home addresses. A member process that agrees to participate in the session replies with the addresses of its input queues; these addresses can be dynamic. The address of an input queue for a mobile process can be on a different processor than its home address. In phase 2, processes bind their output queues to input queue addresses returned in phase 1. For example, in Figure 7, the calendar processes could reside on different machines from their home addresses; when an initiator attempts to set up a session, it performs this two-phase protocol to locate processes and commit them to the session (*initiate-and-commit*).

The present design requires a process to be immobile during its participation in a session: it cannot change the addresses of its input queues during a session though it can change the addresses after the end of one session and before the start of the next. The design can be extended, by using message indirection for instance, to deal with mobility during a session.

2.2 Putting the Pieces Together: A BOF Scheduling Session

Consider the example from Section 1.1 of a secretary setting up a BOF meeting with members from different sites. Prior to the session, each committee member has installed a calendar process on her or his machine. Each calendar process operates within a single address space, communicates with files by standard I/O operations, and communicates with other calendar processes through communication requests. For the actual implementation, an Internet address is associated with each process.

A session is an instance of an application, implemented as a network of processes. As illustrated in Figure 7, the BOF scheduling session consists of many different types of processes: an initiator process that sets up connections and relays address information, user calendar processes with access to the appointment calendars of individual users, and a secretary process that coordinates the collection of information and the decision and broadcasting of a suitable meeting time. Programs corresponding to each process type are installed on the appropriate machines; for the session in Figure 7, the calendar user processes and secretary process are processes running on their respective users' desktop computers.

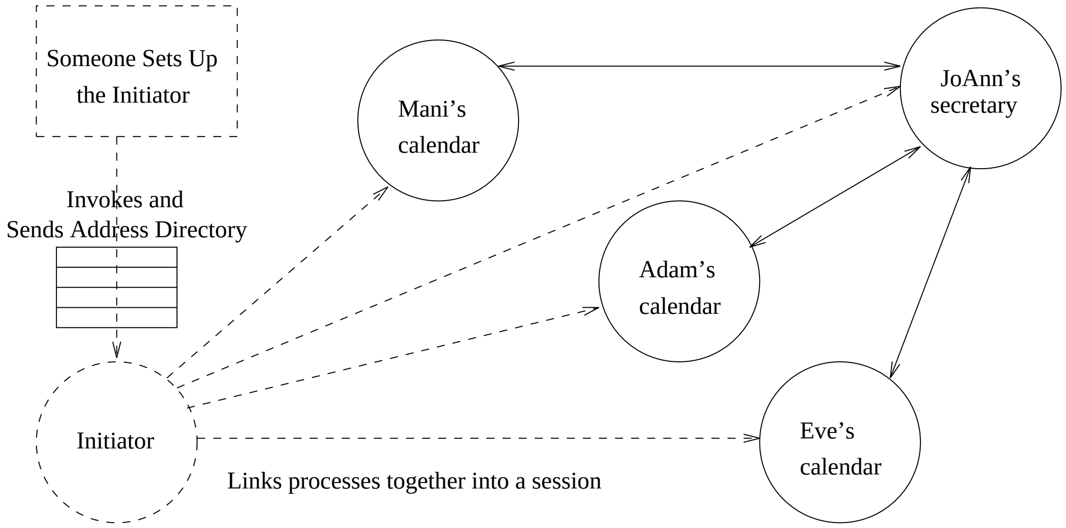

Figure 7: An initiator process uses the invoker's address directory to set up a session between existing calendar and secretary processes.

Associated with each session is an initial process — the *initiator* process — that is responsible for linking processes together. In the BOF scheduling example, someone (e.g., a person or a person's process) sets up the initiator process. Processes are composed in parallel to form a session in two phases, as follows.

The initiator sends a request to each of the processes in the session's initial membership list; this request is a message asking the recipient to participate in the session. Each session has a unique identity: (process number, sequence number). A member process responds to the request either by refusing to participate or by agreeing to participate. It may refuse to participate because (for instance) its access control does not permit this participation, or because it is already participating in another session, and that session's specification forbids the process from concurrently participating in more than one session. If it agrees to participate, it replies with the (global) addresses of its input queues that are to be connected to the output queues of other processes in the session.

After receiving replies from all member processes (or timing out), the initiator sends a second message to all of the members, informing them either to initiate or to abort the session. A message to initiate the session contains the addresses of the input queues to which each member process is to bind its output queues. A process, on receiving the initiate message, binds its output queues to appropriate input queues and starts its threads, and thus begins its participation in the session. After completing their tasks, the member processes close the session, having each modified their local states.

3 Process and Session Specifications and Compositions

Any process on the Internet may attempt to initiate a session. The participants in a session are not known until the session is initiated. Recall that these characteristics originally caused us to label this distributed system *anarchic* in Section 1.1. In this section, we propose compositional methodologies that help application developers deal with the anarchy.

Specifications. A *specification* is a precise definition of behavior. In our model, common specifications allow application developers to write programs with an understanding of:

1. How the processes modify their local states through the *process specifications*.
2. How those processes communicate and interact through the *interface specifications* of the input queues and output queues.
3. How those processes can be composed on a network through the *session specifications*.

We discuss specifications further in Section 4.

Reasoning. Specifications allow application developers to reason about the correctness of their processes, interactions, and sessions. Correctness verification is achieved using *preconditions* and *postconditions*, which are assertions on the states of program components before and after the execution of statements that cause a transition from one state to another.

Composition. Program components can be composed in a number of ways: sequentially, by choice, and in parallel. Given some number of components, with *sequential composition*, all of the components are executed in order, one after the other. By contrast, given some number of components, with *parallel composition*, all of the components are executed in some order that cannot be predicted; this execution might happen concurrently on multiple machines. Also, given a number of alternative components, *choice composition* chooses one to execute under some specified arbitration policy. The different types of composition may be nested to create larger programs from smaller program components.

3.1 Specification and Reasoning about Processes

In our implementation, a process is a Java program with files (i.e., persistent state), that can interact with other processes by sending and receiving messages through its output and input queues. We reason about a process as a state transition system. The state of a process includes the states of its input and output queues. There are two kinds of state transitions in the process: (a) transitions in which the process takes a step, and (b) transitions in which the communication layer takes a step and modifies message queues or raises message-related exceptions.

A state transition in which the process takes a step is an action by a thread of the process. An action can change the program counter of the thread, change local variables, append a message to an output queue, receive a message from an input queue, or query an input queue. An action in which the communication layer takes a step can append a message to an input queue, remove the message at the head of an output queue, or raise message-related exceptions.

A specification of a process is defined in terms of the externally visible aspects of the process: the messages delivered to and sent by the process, and also the process state. For instance, in a calendar application, a specification for processing a "make appointment" message is that the state (e.g., the appointments schedule) has changed appropriately. Even if the entire process state is not externally visible, some predicates on the state (which can be defined as "thought" or auxiliary variables) are visible.

Process specifications are given in terms of safety properties (e.g., *next*, *stable*, and *invariant*) and progress properties (defined using *leads-to*) [CM88, Lam94, CS95]. Processes can only be composed in parallel; we do not deal with sequential or choice composition of processes, though we do support sequential and choice composition of sessions.

3.2 Specification and Reasoning about Sessions

A session is defined in terms of preconditions and postconditions on the states of processes that participate in the session. A session that is initiated with the prescribed preconditions on specified processes must terminate establishing the prescribed postconditions on the processes. For instance, a session to establish a time for a BOF meeting has the precondition *true* and the postcondition that the state (i.e., calendar) of each member attending the meeting is changed to record the appointment for the meeting.

A session may be implemented by parallel composition of processes. Since a session does not have input message queues and output message queues, there is no way to bind one session to another by binding message queues. Since sessions themselves cannot be composed in parallel, sessions can be defined in terms of preconditions and postconditions.

We reason about a session as an atomic operation that can change the states of several processes. Sessions can be composed in any of the ways in which statements in a process are composed. For instance, sessions can be composed using sequential and choice composition. Different threads of a process can execute sessions concurrently. Thus, since our system supports threads (because Java does), parallel composition of sessions is possible by having parallel threads initiate sessions.

Since a session can be encapsulated within a statement in a thread of a process, and since processes can be composed to form sessions, we conjecture that the arbitrary nesting of processes and sessions can be supported, given certain constraints. We are presently investigating what those constraints should be.

These concepts are illustrated in Figure 8. Suppose we have threads running in four Java processes P, Q, R, and S. Using the two-phase initiate-and-commit (described in Section 2.1), processes P, Q, and S can synchronize and enter a session together. The corresponding threads in those processes suspend while the session takes place. Meanwhile, other threads in those processes (and threads in other processes such as R) can execute normally (or enter into other sessions that do not interfere with this session with respect to the corresponding process states). When the session terminates, the threads in processes P, Q, and S resume. Later, if the threads in processes Q and R want to hold a session, they can do so, using the same technique. As indicated previously, we can reason about each of these sessions simply as a single operation in a thread, that potentially modifies the states of the participating processes.

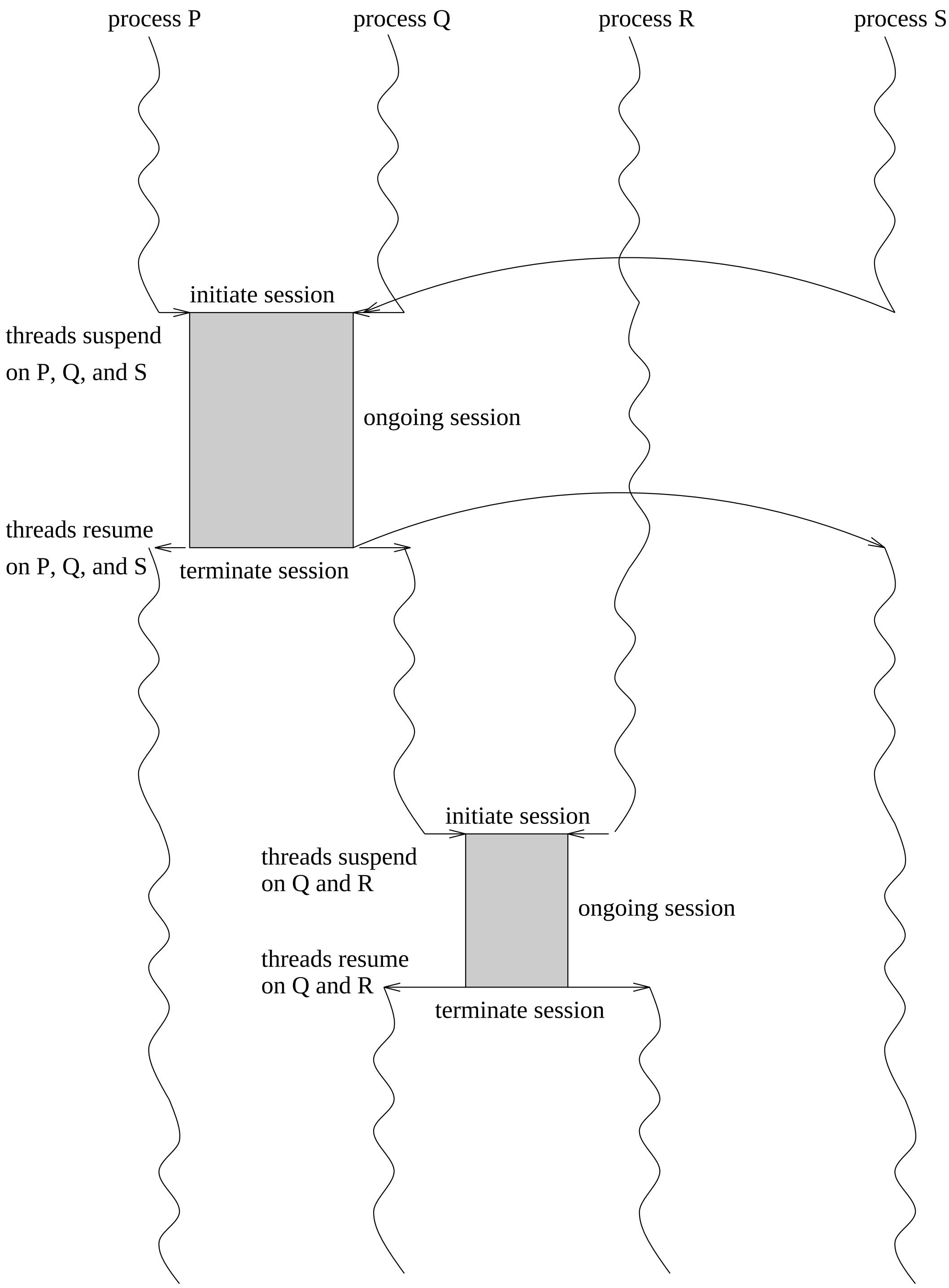

Figure 8: Sessions can be composed into the threads of processes in any of the ways in which other statements can be composed into processes; for example, in this figure, sessions are sequentially composed into process threads. When all of the participating processes commit to a session, the session is initiated, and the corresponding threads suspend; when the session terminates, the modified process states are saved in the persistent store, and the suspended threads resume.

3.3 Services to Support Sessions

Our infrastructure supports services to sessions [CRS+96]. One of the challenges is to provide a service layer that sessions can employ when we do not know in advance what the applications that include the sessions do. Next, we give a brief discussion of some of the service support we have planned for sessions.

Traditional distributed systems (e.g., [Tan95]) are 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 processes is, however, very different from that in traditional systems. We do not expect each process 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 process developers could use in their processes, as needed.

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

We focus our discussion here on inter-process services. Methods for coordination within a process use standard Java classes [SC96]. The questions we address are: How can objects associated with a service be bound into a process in a straightforward way, and, what sorts of services are helpful for process designers?

There are complementary ways of providing services to processes. We can provide a collection of service objects that a designer can include in a process. In addition, we can have a resource manager process, executing on each machine, that provides a rich collection of services to processes 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 process.

Tokens and Capabilities. Distributed operating systems manage indivisible resources shared by processes [Tan95]; we would like to provide service objects with this functionality, which a process 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 processes in a way that avoids deadlock (if processes release all resources before next requesting resources), and detecting deadlock if it does occur (if a process holds on to some resources and then requests more). The designer of a process can separate these service functions from other concerns, and using a library of common service functions can simplify process design.

We treat each resource as a *token*. Tokens are objects that are neither created nor destroyed; once they are initially set up, 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 processes in a session. A token is either held by a process or by the network of token managers. A token manager associated with a process has a data member that maintains the number of tokens of each color that the process holds.

A process can carry out the following operations on its token manager. A *token request* suspends until the tokens requested (i.e., a specified number for each color) are available, and then these tokens are removed from the token manager collection and given to the process. 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. A *token release* releases the specified tokens from the process and returns them to the token managers. If the tokens specified are not actually held, an exception is raised. A process can also probe the total number of tokens of all colors in the system.

The process 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. A process that needs to access the object requests the appropriate token from the token management servlet.

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 process writes the object only if it has all tokens associated with the object, and a process reads the object only if it has at least one token associated with the object.

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 processes release all resources before requesting resources, and release all resouces within finite time, then all requests will be satisfied.

The problem is that processes 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 [CL85]. The global snapshot criterion is satisfied, provided every message that is sent when the sender's clock is T, is received when the receiver's clock exceeds T. A simple algorithm [Lam78] 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. Process designers can separate the generic concerns of clock synchronization from other concerns specific to their application.

Other Servlets. Other servlets for sessions we are investigating include a library to enable the creation and maintenance of distributed data structures (e.g., for diffusing computations), and the infrastructure to permit constrained forms of process stack layering.

Also, some servlets, rather than providing services to sessions, provide services to processes or threads. For example, Java provides constructs for synchronizing threads within a process by using something like a monitor [GYtJT96]. We have implemented and verified other kinds of synchronization constructs — barriers, single-assignment variables, channels, and semaphores — for threads within a process [SC96]. We are extending these designs to allow synchronizations between threads in different processes in different address spaces.

4 Finding Distributed Objects

A collaborative application can require two kinds of searches: finding processes and finding process types. The secretary trying to set up a BOF meeting may have to find (the actual addresses of) a member's calendar processes so that they can be composed into a session. Also, a new member may want to find a calendar process type to set up a process to manage her or his calendar.

To find a specific process, we use standard Web technology: the process is found by looking up the appropriate home page. For instance, the address of Adam's calendar process address is found at Adam's home page.

Finding an appropriate process type is a more interesting problem for several reasons: (a) the specification of the type must be clear so that a potential user can determine if that process type can be composed with the process types of other people with whom that person collaborates, and (b) if the interfaces of two types do not match, some scheme should be employed to refine interfaces to achieve a match. We use the notion of *inheritance* from object technology to solve this problem.

Inheritance. The specification and implementation of a process type is placed in a Universal Resource Locator (*URL*). Inheritance of process types is implemented on top of URLs, as illustrated in Figure 9. First, we discuss location of process specifications and implementations.

The URL that contains a process specification and implementation includes the following parts, defined bottom up, starting from objects and going to sessions:

1. Specifications and implementations of objects that can be sent as messages.
2. Marshaling methods, i.e., methods for packing these objects into byte streams for transportation across a network, and then unpacking the byte streams back into the objects.
3. Specifications of the messages themselves. This includes methods for appending a message of the specified type to an output queue.

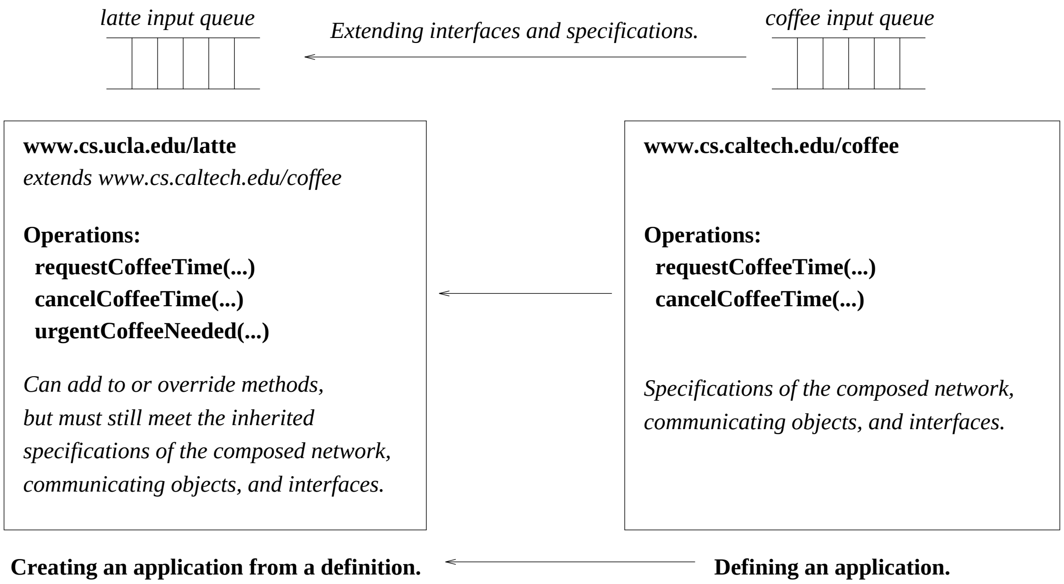

Figure 9: Inheritance types are compatible only if the interfaces are inherited from the same URL (Universal Resource Locator). In this sense, we exploit existing Web technology.

4. Specifications of the types of input and output queues, including the skeleton of the program segment that retrieves a message from an input queue and then deciphers the message, i.e., retrieves its component objects.

5. Specification of the process, in terms of a state transition system. This specification includes the specification of the state changes and messages sent in response to each message received, and details about the implementation of the process itself (so that the process can be installed anywhere). The process specification also defines the binding of the files (i.e., persistent store) to the program.

6. Specification of the sessions in which the processes can be used.

A process type can be tailored to a specific group of users by refining its type; in object-oriented terms, we create a subtype that inherits from the parent type, as illustrated in Figure 9. The subtype must, of course, support the objects, messages, and sessions of the parent type; it may also support additional objects, messages and session-interactions.

Our initial design for implementing inheritance is by brute force: a process type can be stated as inheriting from another type, but no tools are provided to check that one type is a subtype of another — instead, it is merely an assertion provided by the programmer. The URL that contains the subtype's specification and implementation contains pointers to its parent types. Processes of types P_t and Q_t can be composed in a session C if both types are the same or one is a subtype of the other, and both support sessions of type C. Thus, determining whether two interfaces (objects, messages, message-queues, or processes) "match" is done by following the pointers in their URLs. A more sophisticated mechanism could be used, but this is not the focus of our research.

Composing Distributed Objects. The composition of distributed objects requires checking if their interfaces match; once that checking is complete, a session connection can be set up, and a series of transactions can occur. For example, as shown in Figure 10, independently running distributed processes can collaborate on-the-fly over the Internet safely, because their interfaces have been verified to be appropriate.

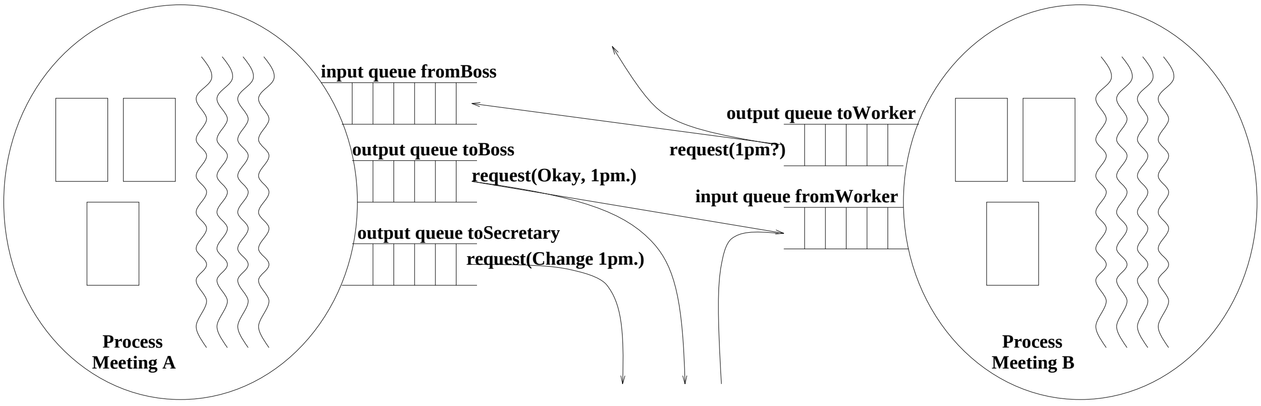

Figure 10: Sessions can be set up on-the-fly once interface matches have been checked. Here, a boss can have her meeting process request a 1pm get-together with her workers, and the workers' processes can receive that request and (automatically or interactively) reply, also requesting to their secretaries to change their 1pm appointments, if needed.

Mobility and Resource Discovery. We are investigating enhancements in our system design to support communication between mobile objects through the use of directory registries [CS96]. As described at the end of Section 2.1, each process can have an immutable home address URL (that is available to Web search engines); this home address includes the unchanging address of an input queue to which requests to participate in sessions are sent. As shown in Figure 11, processes can use these home URLs to set up sessions, as the input queue for a mobile process can be on a different processor than its home address.

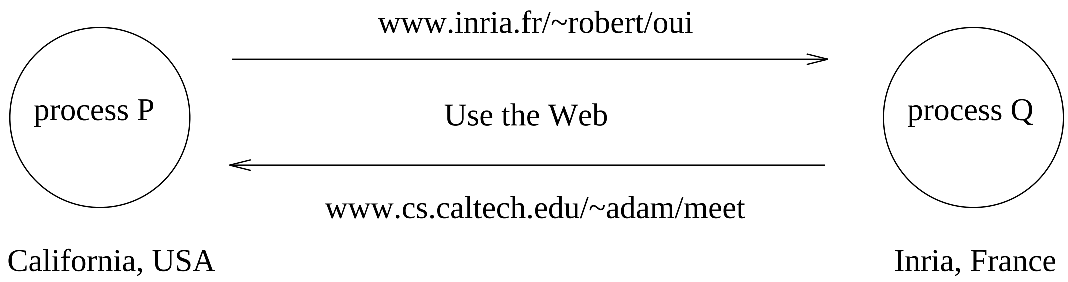

Figure 11: The Web can be used to locate the interfaces and specifications of distributed (and possibly mobile) processes. The Web can also be used to locate the running programs themselves.

5 Related Work

Research has yielded many theories, methods, and tools to help application developers use distributed systems [Tan95], distributed languages [BST89], and distributed algorithms [CM88, Lyn96]. Our programming model and theories of structured composition build on this research and recent work in formal methods [DS90, CT92, AL93, Cha94, Lam94, CS95].

To supplement our model, we are designing and implementing a communication infrastructure [CRS+96], in which processes can be written as multithreaded Java objects called *dapplets*. Using the compositional theory described in this paper, dapplets can be composed into sessions, wherein the states of the component dapplets can be modified in a peer-to-peer fashion through transactions. We provide formally verified reliable libraries for synchronization between threads (e.g., single-assignment variables, reusable barriers, locks, and semaphores, as specified in [SC96]), and will be working on formally verified reliable libraries providing services for use in sessions (e.g., tokens, clocks, distributed data structures, and stack layering facilities, as specified in [CRS+96]).

Our implementation shares many design features of *network objects* [BNOW95], including distributed typechecking, transparent remote invocation, marshaling, and buffered streams. A network object is an object whose methods can be invoked by other local and remote programs; network objects ensure distributed type safety with the *narrowest surrogate rule*, which allows programmers to export new versions of distributed services as subtypes of previous versions. Many systems, including the *Obliq* distributed scripting language [Car95], have been built using Modula-3 network objects. Obliq objects have state and are local to a site; Obliq enables a dynamic form of distributed programming, where objects can redirect their behavior over the network, and where computations can roam between network sites.

Obliq allows mobility of program code as well as the context in which the code operates; similarly, *Telescript* [Whi94] allows mobile agents that carry their context with them as they move from location to location. Whereas Obliq contexts can include established network connections, *agents* are self-contained and resource-limited: instead of communicating remotely with other locations, agents move themselves to a remote location site and communicate locally. Agents share collaborative characteristics with our dapplets: they can run unattended for a long time, meeting and interacting with other agents.

Although our dapplet support for collaborative distributed application development was implemented using Java, the theories and tools for composition we propose are employable in conjunction with other platforms, such as CORBA-compliant Object Request Brokers [Obj95]. *CORBA* is a language-independent industry standard for remote invocation; through *Object Request Brokers*, objects in one location in a network can invoke methods on other objects in the network in a location-independent manner. This characteristic is best suited for client-server application development; however, the structured compositional approach described in this paper can also be a useful design methodology when developing CORBA-compliant distributed peer-to-peer object computations.

Putting the concepts discussed in this paper into the distributed object context, processes are objects that interact using remote procedure calls [BN84], and the interfaces through which they receive messages are the public interfaces they export. Sessions are conglomerations of interacting objects; such object interface definitions are part of the CORBA standard, which defines an implementation language-independent interface definition language. This interface definition provides a convenient framework in which to specify the behavior of services available to sessions as well.

An example of a CORBA-like object system is the Inter-Language Unification (*ILU*) system [Gro96]. The object interfaces provided by ILU hide implementation distinctions between different languages, between different address spaces, and between operating system types. ILU can be used to build multi-lingual distributed object systems; remote procedure call services can be described and used as ILU objects.

Whereas CORBA provides a useful architecture for disseminating structured information, the World Wide Web [BLCGP92] has proven useful as a mechanism for distributing less structured information. As a result, many research groups [OW96] are presently working on systems and applications in which distributed (and possibly mobile) objects interact over the World Wide Web. One interesting approach is the idea of nodes with *scalable intelligence* [Fuc96], in which the objects of a peer-to-peer system dynamically integrate tools into distributed cooperative applications. To support ad hoc collaborations, these objects can employ domain-specific little languages to handle interactions, and these languages can be understood both by the agent objects themselves and the humans who interact with them. In future work, we plan to investigate this approach and others in the context of specifying, reasoning about the composition of, and implementing our processes and sessions for distributed collaborations.

6 Summary

We presented a modular model for developers of distributed systems where the exact ways in which interacting applications can be composed may be unforeseeable. Our theory provides two fundamental structuring units, *processes* and *sessions*, that can be developed using nested sequential, choice, and parallel composition. We investigated solutions to the problems of specifying, composing, reasoning about, and implementing distributed applications, through the use of processes and sessions. Combined with the theory of systematic process and session composition, our infrastructure tools, implemented using Java, allow a developer to create collaborative distributed applications on the Internet.

References

[AL93] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993. Also available as DEC SRC Research Report 66.

[BLCGP92] T. Berners-Lee, R. Cailliau, J. Groff, and B. Pollermann. World Wide Web: The Information Universe. *Electronic Networking: Research, Applications, and Policy*, 1(2), 1992.

[BN84] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BNOW95] A. Birrell, G. Nelson, S. Owicki, and E.P. Wobber. Network Objects. *Software Practice and Experience*, 25(S4):87–130, December 1995. Also available as DEC SRC Research Report 115.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, January 1995.

[Cha94] K.M. Chandy. Properties of Concurrent Programs. *Formal Aspects of Computing*, 6(6):607–619, 1994.

[CL85] 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.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[CRS+96] K.M. Chandy, A. Rifkin, P.A.G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth Workshop on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[CS95] K.M. Chandy and B.A. Sanders. Predicate Transformers for Reasoning about Concurrent Computation. *Science of Computer Programming*, 24(2):129–147, April 1995.

[CS96] K.M. Chandy and E.M. Schooler. Designing Directories in Distributed Systems: A Systematic Framework. In *Proceedings of the Fifth Workshop on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[CT92] K.M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, MA, 1992.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, New York, 1990.

[Fuc96] M. Fuchs. *Let's Talk: Extending the Web to Support Collaboration.* Walt Disney Imagineering, 1996.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley Developers Press, Sunsoft Java Series, 1996.

[Gro96] ILU Group. *Inter-Language Unification.* Xerox Parc, Palo Alto, CA, 1996.

[GYtJT96] J. Gosling, F. Yellin, and the Java Team. *The Java Application Programming Interface.* Addison-Wesley Developers Press, Sunsoft Java Series, 1996.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. Also available as DEC SRC Research Report 79.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, San Francisco, CA, 1996.

[Mas97] B.L. Massingill. *Parallel Programming Archetypes in Scientific Computing*. PhD thesis, California Institute of Technology, 1997.

[Obj95] Object Management Group. *The Common Object Request Broker: Architecture and Specification (CORBA)*. OMG, 1995. Revision 2.0.

[OW96] Object Management Group and World Wide Web Consortium. *Proceedings of the Workshop on Distributed Objects and Mobile Code*. OMG/W3C, Boston, MA, June 1996.

[Pos80] J.B. Postel. *User Datagram Protocol*. RFC 768, August 1980.

[Rif96] A. Rifkin. *Application Development using Analytic and Experimental Performance Tuning*. Technical Report CS-TR-96-09, Computer Science Department, California Institute of Technology, 1996.

[SC96] P.A.G. Sivilotti and K.M. Chandy. Toward High Confidence Distributed Programming with Java: Reliable Thread Libraries. In *International Conference on Software Engineering*, July 1996.

[Ste94] W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA, 1994.

[Tan95] A.S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Tho96] J. Thornley. *A Parallel Programming Model with Sequential Semantics*. PhD thesis, California Institute of Technology, 1996.

[Whi94] J.E. White. *Telescript Technology: The Foundation for the Electronic Marketplace*. General Magic, Inc, 1994.