Journal of Supercomputing, 11, 1–18 (1997)

Webs of Archived Distributed Computations for Asynchronous Collaboration*

K. MANI CHANDY mani@cs.caltech.edu
JOSEPH KINIRY kiniry@cs.caltech.edu
ADAM RIFKIN adam@cs.caltech.edu
DANIEL ZIMMERMAN dmz@cs.caltech.edu
Department of Computer Science 256-80, California Institute of Technology, Pasadena, California 91125 USA; http://www.infospheres.caltech.edu/

Received August 8, 1996 ; Revised April 15, 1997
Editor: Salim Hariri

Abstract. We identify the mechanisms needed to construct archivable webs of distributed asynchronous collaborations and experiments. The distinguishing feature of our approach is that the component tools, software, data, and even participants are distributed over a worldwide network. We perform a requirements analysis of an infrastructure that supports such applications, and present the Caltech Infospheres Infrastructure as a prototype that satisfies the requirements identified. In describing this prototype, we highlight the useful mechanisms provided, present an algorithm for using the Infospheres Infrastructure to perform asynchronous global snapshots for archiving, and suggest future areas of exploration.

Keywords: distributed systems, archiving, global snapshots, asynchronous collaboration, world wide web, computational experiments, infospheres, component technologies, composition

1. A Vision: Archived Distributed Computational Experiments

Since its creation, the Internet has been used for information sharing and collaboration. In this paper, we describe the design of a software technology that allows any component of a distributed system to (i) archive a "global snapshot" of the distributed system, (ii) record events within components of the system, and (iii) replay a distributed computation by resurrecting the system from an archived global snapshot and executing the archived events from the snapshot onward. An annotated collection of archived global snapshots, events, and documents can be linked into the World Wide Web automatically, allowing distributed systems to be restarted from their saved states, see these computations unfold, and follow the links to related computations.

The idea of archiving states and replaying events has been employed previously in such contexts as data backup, compiler analysis (Mellor-Crummey, 1992), and application debugging. Our contribution is that of exploring methods for, and

* The Caltech Infospheres Project is sponsored by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244, by the CISE directorate of the National Science Foundation under Problem Solving Environments grant CCR-9527130, by the NSF Center for Research on Parallel Computation under Cooperative Agreement Number CCR-9120008, and by Novell, Inc.

identifying potential benefits of, archiving states and replaying events in distributed computations. Specifically, we consider systems composed of autonomous opaque objects with dynamic interfaces distributed across the Internet.

We begin by describing our vision of a web of archived distributed computations: first, we provide an overview of software component technology, and then we discuss some potential applications for the archival of computations in distributed component systems. Since component technology is not the focus of this paper, we have restricted our discussion of it to the details relevant to the archival of distributed computations.

1.1. *Component Technology*

Component technology focuses on the representation and use of self-contained software packages which can be *composed*, or attached together, to create complete applications. Each component has an *interface* that specifies the compositional properties, and sometimes the behavior, of that component. Components can be composed either through static linking at compile time, or through dynamic linking over a network at run time. Our focus is on systematic composition of components that have dynamic interfaces and use asynchronous messages (Chandy et al., 1996). There are several popular commercial component technologies, including CORBA (OMG, 1995), OpenDoc (MacBride and Susser, 1996), ActiveX (Chappell, 1996), and Java Beans (Java Beans, 1997).

Software component technology offers the potential for building new applications quickly and reliably. Rapid application development tools for creating component-based software are emerging. However, current component infrastructures are complex, requiring application developers to compose components at compile time using stubs and skeletons(OMG, 1995, Java RMI, 1997). Our focus is on dynamic composition of components at run time and methods for reasoning about the behavior of the resulting "collective" applications.

As an example of a collaboration-based distributed component system, imagine a group of researchers and observers working together on an experiment with several components:

- data sets from databases in Houston and Syracuse;
- a program composition tool at Caltech;
- a CFD solver on a supercomputer at Argonne;
- solid-mechanics simulators on a network of workstations at Los Alamos;
- visualization engines in the offices of the researchers; and
- a classroom of students several weeks later, using standard web software to review the experiment and discuss it with their professor.

Trends in compositional tool and network connectivity technologies suggest that such examples will become feasible. In this section, we discuss the software technology needed to create such a distributed experiment: opaque components with dynamic interfaces, selected from a worldwide pool of components capable of collaboration, that can record their states throughout the collaboration.

1.1.1. Opaque Distributed Software Components. An *opaque* (or "black box") component furnishes a programmer only with its interface specifications, not its actual implementation. The internal structure and behavior of an opaque component are completely hidden from other components. We assume that the components participating in a distributed collaboration of the type described above will be opaque, because it is unreasonable to require that experimenters have access to the internal workings of the components they use.

The opacity of components implies that the procedure for archiving distributed state must itself be distributed. Since no component has access to the implementation of any other component, no single component can archive the state, or even a state description, of another component in the system. Therefore, each component must record its own state and archive it locally, and archived states of the entire system must be obtained by combining the locally archived states of the individual components.

1.1.2. Dynamic Interfaces and Dynamic Composition. Component interfaces can range in dynamism from completely static to completely dynamic. Most component systems with communication based on remote procedure calls (such as CORBA, Java RMI, and Microsoft COM and DCOM) support the use of static interfaces, which can be type checked at compile time. However, there are problems associated with the use of static interfaces in dynamic distributed environments. Components with dynamic interfaces can interact more successfully in such environments but, since the syntax of their interactions cannot be checked at compile time, the components must handle faulty communication links and unexpected interface changes at run time. We have developed a prototype infrastructure that supports composition of components with dynamic interfaces (Chandy et al., 1997).

In this paper, we consider components with dynamic interfaces in a dynamic environment, though the central ideas relating to archiving distributed computations are applicable to components with static interfaces as well. The relevance of dynamic interfaces to the archival of distributed computations is that the state of a component must include its interface. For example, if the interface of a component is defined in terms of communication channels, and the number and types of those channels can change during a computation, then the archived state of the component must include information describing the channels in addition to any other information needed by the component.

1.1.3. Selecting Components from a Worldwide Pool. Ideally, scientists should be able to develop an application by using components selected from a worldwide pool. These components may be located at different sites, may be running on systems with various architectures and operating systems, and might have restricted availability.

Suppose, for instance, that an aeronautical engineer wants to do a multidisciplinary optimization experiment on airfoils. This experiment requires the composition of a solid mechanics computation dealing with vibrations and a fluid dynamics computation dealing with airflow. Many sites might offer a component that performs fluid dynamics computations, but these sites might differ in computation capability, access restrictions, and cost. The engineer should be able to select whichever component fits his needs, whether it is at Caltech, Los Alamos, or San Diego. Our vision is that it should be possible to develop an application by using components at different remote sites as easily as by using only local resources.

A worldwide pool of components is relevant to the archival of distributed computations because of scaling considerations. If all the components were located on an intranet serving a small, single-site campus, then a potential solution would be to take a global snapshot of the entire intranet. However, since the Internet has many autonomous units, such an approach is not feasible on a global scale. Also, the use of resources at multiple distributed sites raises issues of security, resource allocation, and privacy. We will not focus on these issues in this paper, since some solutions to these problems exist, such as Java's sandbox model (Gosling, Joy, and Steele, 1996) and ActiveX's code signing model (Chappell, 1996).

1.1.4. Modes of Collaboration. There are two types of collaboration between groups of people using programs, control devices, and measuring instruments:

- *synchronous* collaboration occurs when all components collaborate at the same time, usually requiring the continual presence of human beings.

- *asynchronous* collaboration occurs when components can participate at different times over the course of a collaboration, only occasionally requiring the presence of human beings.

Teleconferencing and multi-user whiteboarding are examples of synchronous collaboration; these interactions are typically carried out by small groups of people for durations on the order of minutes to hours. A concurrent version-control system, with people working together on documents over extended periods of time, exemplifies asynchronous collaboration. In such a system, different annotated copies of documents flow through the system as individuals check in their work and update their workspaces. In this paper, we consider methods of archiving distributed system states for asynchronous collaboration, though the ideas can be used in synchronous collaborations as well.

1.2. *A World Wide Web of Archived Distributed Experiments*

Standard Web technologies can be used to annotate and link the archived states of distributed computations. Components can automatically generate the proper representation of their states and cross-reference results with related experiments. Such a system enables an experimenter to follow the web of links and recreate past experiments in an order that is meaningful to her. The expansion of the concept of a "Web document" to include an archived distributed system allows a participant to recreate an archived distributed experiment as easily as she can bring up a document, by clicking on a link on her Web browser.

We postpone discussion of methods for taking global snapshots and replaying the events of distributed systems, and how to use the World Wide Web for archiving distributed experiments, to section 3.5. For now, we explore potential applications of this technology.

1.2.1. Application: Computational Science. Consider a collaboration in which researchers at different sites work together on a computational experiment that requires the devices and software at different locations to be composed together. Now consider a scientist, Dana, who joins the team after this particular experiment and would like to repeat the experiment to reconstruct the sequence of events that generated its final results. To do so, she must have access to an archive of all the tools used to conduct the experiment, including all the input and output data and all the annotations added by the participants.

Ideally, Dana should enter the archived virtual laboratory and find it exactly as it was when the experiment was conducted. This gives her the option of either conducting a modification of the experiment herself or witnessing the original experiment as it unfolds before her. As Dana explores the research team history, she can use the Web to view shared documents and other archived distributed experiments.

An important aspect of this archived virtual laboratory is that the components of the laboratory are geographically distributed: input data sets are generated at one site, a meshing computation is conducted at a different site, and the output data is post-processed prior to visualization at a third site. What is being archived is a distributed system consisting of components from all three sites.

After Dana explores one archived experiment, she may study the annotations of the experimenters and then follow links to related experiments. She can follow links to later experiments by the same collection of experimenters, explore attempts by other groups to replicate the experiments with different tools, or jump to a document discussing the public policy issues raised by the experiments. She can even reuse data or components from a previous experiment in a new experiment of her own design.

1.2.2. Application: Maintenance of Large Distributed Systems. Consider several crews of technicians repairing an electric power distribution network after a grid failure. Crews work concurrently at different sites, both in the office and in the field, making the collection of crews a distributed system much like the power network itself. It may be desirable to take global snapshots and log events in these systems, and then link these states into the Web. This archive can then serve as a training tool for crew members, allowing someone joining the repair effort late to understand the current situation, or be used to "roll back" the system state to determine the exact conditions that caused the failure in the first place.

1.2.3. Application: Corporate Technical Support. The Information Technology (IT) division of a large corporation often has responsibility for thousands of systems and tools. This complexity makes for a challenging problem-resolution environment. An archive of distributed computations can serve as a corporate memory. Restoring an archived distributed state and replaying a distributed computation can help in training, problem-resolution, and maintenance.

1.2.4. Application: Distributed Education. The use of global snapshots to record states of a distributed simulation would be extremely useful for educational purposes. For example, if a military combat simulation were recorded in this fashion, the simulation could be replayed for personnel who were not participating to give them the benefit of the experiences of the participants. In addition, such recordings would facilitate detailed tactical analysis, allowing the simulation to be restarted at specific points and the effects of changing specific tactical actions to be examined.

We believe that there are many more applications of this technology than described here, and expect that future research by ourselves and others will investigate these applications in detail. However, such investigations are outside the scope of this paper.

We continue by analyzing the requirements for a distributed infrastructure which supports asynchronous collaborations of the type we have described.

2. Requirements Analysis

As discussed previously, an infrastructure to support applications like the above examples must support the composition of distributed opaque components with dynamic interfaces. These components must be able to participate in both synchronous and asynchronous collaborations. The infrastructure should assist in locating and composing components on the Internet. Finally, it should be possible to archive a distributed experiment and resurrect it with reasonable use of resources, and these archived distributed computations should be linked into the Web just as other documents are.

2.1. *Opaque Distributed Software Components*

The only visible aspects of an opaque component are (i) its external interface, so that other components can connect, and (ii) a specification of the component. In a distributed system, the interface is specified in terms of remote method invocations (Gosling, Joy, and Steele, 1996), object-request brokers (OMG, 1995), or messages (Hoare, 1978, Chandy et al., 1997). Each approach has advantages and disadvantages, but the specific form of the interface is less important than the fact that the component implementations are hidden. The infrastructure must support at least one of these methods of interface specification.

2.2. *Dynamic Interfaces and Interactions*

A component must be able to adapt to changing conditions in a computation. These include the addition of new components to the computation, temporary unavailability of communications resources, and other common situations which arise in Internet-based distributed systems. One way to deal with the dynamic environment is to allow a component to change its interface and connections to other components, during the course of a computation, so we require that the infrastructure allow component interfaces and interconnections to be completely dynamic.

2.3. *Modes of Collaboration*

All components participating in a synchronous collaboration must be active concurrently. By contrast, components participating in an asynchronous collaboration need not be active concurrently; any given component may be quiescent, becoming activated only when a communication arrives for it. The advantage of asynchronous collaboration is that the participating components need not hold resources concurrently, since they use resources only when they are computing. The disadvantage is that handling an incoming communication can be expensive, because the communication must be handled by a daemon that activates the quiescent component and then forwards the communication. Because of this tradeoff, we require the infrastructure to support both synchronous and asynchronous interactions, allowing individual component application developers to choose whichever mode is appropriate for their application.

2.4. *Persistence*

Components must be persistent, because a collaboration involving a set of components may last for years. Rather than forcing a component to stay active for the life of its collaborations, it is advantageous to design the component system such that the life cycle of a component is a sequence of active phases separated by quiescent phases. In such a system, when a component is quiescent, its state is serialized (and can, for example, be stored in a file) and the component uses no computing resources. When a component is active, it executes in a process slot or thread and listens for communications. Components designed in this way are often quiescent for most of their lifetimes, so the fact that quiescent components use no computing resources allows many more components to exist on the same machine than could possibly run simultaneously. The infrastructure must support the storage of persistent state information by individual components. In addition, it is desirable for the infrastructure to provide some method of efficiently updating persistent state information, such as by saving only incremental changes.

2.5. A World Wide Web of Archived Distributed Experiments

Web technologies already provide the necessary mechanisms for linking archived distributed computations so that dynamic content representing the state of a component, distributed experiment, or computation can be viewed, hyperlinked, and indexed for searching. Users can take advantage of web browsers to read web pages and to follow links to archived information. The infrastructure must provide a way to generate such pages automatically, as well as a way to restart a distributed computation from its saved state by clicking on a link in a document.

2.6. Tangential Issues

There are several engineering problems that have little to do with distributed systems but are nevertheless important. If a computer scientist wanted to resurrect von Neumann's experiments with the earliest computers, he could not do so because the machines used by von Neumann no longer exist. Likewise, he could not resurrect collaborations that used old versions of operating systems or other tools.

There are also resource reservation and security issues in such a system, especially if it simultaneously utilizes expensive and rare equipment at multiple sites. If a collaboration requires 256 nodes of a supercomputer, a researcher clicking on the link to resurrect that collaboration will probably have to wait to acquire those nodes. Resurrecting a distributed state may require acquisition of real resources.

Because our focus in this paper is on resurrecting recently archived distributed computations, we are not discussing these important issues here.

3. The Infospheres Infrastructure

In this section, we briefly describe the *Infospheres Infrastructure* (also called the II prototype) (Chandy et al., 1996, Chandy and Rifkin, 1997, Chandy et al., 1997), and show how it satisfies the requirements identified in section 2.

3.1. Infospheres Framework

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

As an illustration of these structuring mechanisms, consider a consortium of research institutions working on a common problem. This consortium has a personal network composed of processes that belong to the infospheres of the consortium members. This personal network provides a structured way to manage the collection of resources, communication channels, and processes used in distributed tasks such as determining meeting times and querying distributed databases. Each session of this personal network handles the acquisition, use, and release of resources, processes, and channels for the life of one specific task.

Infospheres are discussed in detail as part of the user's guide to our framework (Infospheres, 1997). Here, we focus on the conceptual models for processes, personal networks, and sessions.

3.2. Conceptual Model: Processes

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

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

Ready processes occupy process slots and can make use of other resources provided by the operating system. By contrast, processes in the frozen state do not occupy process slots and cannot actively make use of any other resources provided by the operating system. The only resource used by a frozen process is the storage space, such as a small file or a database entry, which holds process state information.

3.2.2. Freezing, Summoning, and Thawing Processes. Each process has a *freeze* method, which saves the state of the process to a persistent store, and a *thaw* method, which restores the process state from the store. A typical process remains in the frozen state nearly all the time, and therefore consumes minimal system resources. In our framework, only waiting processes can be frozen, and they can be frozen only at process-specified points. Except for its persistent store, all system resources held by a process are yielded when its freeze method is invoked.

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

3.2.3. Process Migration. Frozen processes can migrate from one machine to another, but ready processes cannot. This restriction allows ready processes to communicate using our framework's underlying fast transport layer, which requires static addresses for communication resources. All processes have a permanent "home address" from which summons can be forwarded. Once a process becomes ready at a given location, it remains at that location until frozen. While a particular process may be instantiated at any location, its persistent state is always stored at its home address.

3.3. Conceptual Model: Personal Networks

A personal network consists of an arrangement of processes and a set of directed, typed, secure communication channels connecting process output ports to process input ports. Its topology can be represented by a labeled directed graph, where each node is a process and each edge is a communication channel labeled with its type and the input and output ports connected by that channel. Since processes can freely create input ports, output ports, and channels, the topology of a personal network is completely dynamic.

3.3.1. Communication Structures. Processes communicate with each other by passing messages. Each process has a set of *inboxes* and *outboxes*, collectively called *mailboxes*. Every mailbox has an associated type and access control list, both of which are used to enforce personal network structure and security.

A connection is a first-in-first-out, directed, secure, error-free broadcast channel from the outbox to each connected inbox. In our framework, connections are asymmetric: a process can construct a connection from any of its outboxes to any set of inboxes for which it has references, but cannot construct a connection from an outbox belonging to another process to any of its inboxes.

3.3.2. Message Delivery. Our framework's communication layer works by removing the message at the head of a nonempty outbox and appending a copy to each connected inbox. If the communication layer cannot deliver a message, it raises an exception in the sender containing the message, the destination inbox, and the specific error condition. The system uses a sliding window protocol (Peterson and Davie, 1996) to manage the messages in transit.

The communication layer eventually handles every message at the head of an outbox. The conceptual model uses asynchronous messages rather than remote procedure calls, because the range of message latencies across the Internet makes message passing with synchronous remote procedure calls impractical. However, the structure of our communication layer allows us to consider message delivery from an outbox to inboxes as a simple synchronous operation even though the actual implementation is complex and asynchronous.

3.3.3. Dynamic Structures. A process can create, delete, and change its mailboxes, in addition to (as mentioned above) being able to create and delete connections between its outboxes and other processes' inboxes. The operation of creating a mailbox returns a global reference to that mailbox that can then be passed in messages to other processes. Since a process can change its connections and mailboxes, the topology of a personal network can evolve over time as required to perform new tasks.

When a process is frozen, all references to its mailboxes become invalid. This invalidation of mailbox references allows frozen processes to move and then be thawed, at which point the references to its mailboxes can be refreshed via a summons.

3.4. Conceptual Model: Sessions

A session encapsulates a task carried out by (the processes in) a personal network (Chandy and Rifkin, 1997). It is *initiated* by some process in the personal network, and is *completed* when the task has been accomplished. A later session with the same processes may carry out another task. Thus, a personal network consists of a group of processes in a specified topology, interacting in sessions to perform tasks.

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

1. As long as any process within the session holds a reference to a mailbox belonging to another process within the session, that reference must remain valid.

2. A mailbox's access control list cannot be constricted as long as any other process in the session holds a reference to that mailbox.

The session constraint ensures that, during a session, information flows correctly between processes. An important corollary to the session constraint is that, because no valid references to their mailboxes exist, frozen processes cannot participate in sessions.

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

There are many ways of satisfying the session constraint. One simple way is to ensure that every process participating in a given session remains ready until that session terminates, and that once a process sends a given mailbox reference to another process in the session it leaves that mailbox unchanged for the duration of the session. Another approach is to have the initiating process detect the completion of the task using a diffusing computation or other common termination detection algorithm, after which it can inform the other session members that the session can safely be disbanded.

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

3.4.3. Communication Within Sessions. During a session, it is vital that the processes receive the quality of service required to accomplish their task. Therefore, communication is routed directly from process to process, rather than through object request brokers or intermediate processes as in client-server systems. Once a session is constructed, our framework's only communication role is to choose the appropriate protocols and channels. A session can negotiate with the underlying communication layer to determine the most appropriate process-to-process mechanism. While the current framework supports only UDP, we plan in future releases to support a range of protocols such as TCP and communication layers such as Globus (Foster and Kesselman, 1996).

3.5. Archiving Distributed States

We have now described our prototype software infrastructure; next, we describe an algorithm that can be used by the infrastructure to archive distributed states. This is a variant of the global snapshot algorithm (Chandy and Lamport, 1985) in which a clock, or sequence number, is stored with the snapshot state. Within the snapshots, these logical clocks can be used for timestamping (Lamport, 1978).

3.5.1. The Global Snapshot Algorithm. If all components recorded their complete states (including the states of their mailboxes) at a specified time T, then the collection of component states would be the state of the distributed system at time

T. The problem is that the clocks of the components can drift and, as illustrated by the following example, even a small drift can cause problems.

Two components P and Q share an indivisible token that they pass back and forth between them. P's clock is slightly faster than Q's clock. Both processes record their states when their clocks reach a predetermined time T. Assume that the token is at Q when P's clock reaches T, so so P's recorded state shows that P does not have the token. Then, after Q has sent the token to P, Q's clock reaches time T. Q's recorded state then shows that Q does not have the token. Therefore, the recorded system state — the combined recorded states of P and Q that shows that no token is anywhere in the system — is erroneous. The basic problem arises because Q sends a message to P after P records its state but before Q records its state.

We describe our algorithm in terms of taking a single global snapshot. In practice, we will need to take a sequence of global snapshots, and extending the single snapshot algorithm to take sequences of snapshots is straightforward.

Initially, some component records its state; the mechanism that triggers this initial recording is irrelevant. Perhaps a component records its state when its local clock gets to some predetermined time T, and the component with the clock that reaches T first is the first to record its state.

Each message sent by a component is tagged with a single boolean which identifies the message as being either (i) sent before the component recorded its local state, or (ii) sent after the component recorded its local state. In our infrastructure, every message is acknowledged, so each acknowledgment is also tagged with a boolean indicating whether the acknowledgment was sent before or after the component recorded its state. When a message tagged as being sent after the sender recorded its state arrives at a receiver that has not recorded its state, the infrastructure causes the receiver's state to be recorded before delivering the message. Acknowledgements are also tagged, and are handled in the same way. Thus, the algorithm maintains the invariant that a message or acknowledgment sent after a component records its state is only delivered to components that have also recorded their states.

The issue of acknowledgments is somewhat subtle, so we describe it in more detail. Consider a component P sending a message m to a component Q. The message m is at the head of an outbox of P. The message-passing layer sends a copy of m to Q's inbox, to which that outbox is connected. Note that m remains in the outbox while the copy of m is in transit to Q's inbox. When the acknowledgment for m arrives at P, then and only then is message m discarded from P's outbox. If the acknowledgment is a post-recording acknowledgment, then P's state is recorded before the acknowledgment is delivered, and therefore P's state is recorded as still having message m in its outbox.

3.5.2. Repeated Snapshots. The algorithm for taking a single snapshot of an entire distributed system requires each component to have a boolean indicating whether that component has recorded its state. Also, each message and acknowledgment has a boolean field indicating whether that message or acknowledgment was sent before or after the sender of that message or acknowledgment had recorded its state. For repeated snapshots, the boolean is replaced by a date represented by a sequence of integers for year, month, day, time in hours, minutes, seconds, milliseconds, and so on, to the appropriate granularity level. The date field of a component indicates when the component last recorded its state, and this date field is copied into messages and acknowledgments sent by the component. If a component receives a message or acknowledgment with a date that is later than its current date field, it takes a local snapshot, updates its date field to the date of the incoming message, and (if necessary) moves its clock forward to exceed the date of the incoming message.

3.5.3. Replaying a Distributed Computation. There is a distinction between having the saved state of a distributed computation and being able to replay the computation. An archived snapshot helps in a variety of ways but, because some distributed computations are nondeterministic, it does not guarantee that the distributed computation can be replayed.

Our components are black boxes, so we cannot tell whether a component is deterministic. Re-executing the computation of a nondeterministic component from a saved state can result in a different computation, even though the component receives a sequence of messages identical to the sequence it received in the original computation. Replaying precisely the same sequence of events requires each component to execute events in exactly the same order as in the original sequence, so the replay has to be deterministic. For example, if there is a race condition in the original computation, then the replay must ensure that the race condition is won by the same event as in the original. Since components are black boxes, the II cannot control events within a component. Therefore, we rely on the designers of the components to have a record-replay mechanism for recording the event that occurs in each nondeterministic situation and playing back this event correctly during replay.

During replay, the II ensures that messages are delivered to a component in the same order as in the original computation, provided all components in the computation send the same sequences of messages. If the components have deterministic replay, the computation from the saved state will be an exact replay: a sequence of events identical to those of the original computation.

The II guarantees that messages are delivered in the same order as in the original computation in the following way: a *mail daemon* executes on each computer that hosts components, logging the outbox, inbox and message id for each incoming message. Because the contents of the messages are not necessary to properly deal with nondeterminism in the message-passing layer, they are not recorded by the mail daemon. During replay, the mail daemon holds messages that arrive in a different order, delivering them to the appropriate inboxes only after all previous messages in the original computation have been delivered.

3.5.4. A World Wide Web of Distributed Spaces. The existing Infospheres Infrastructure supports saving the states of components and summoning components from these archived states to form new sessions. When a component is summoned from an archived state, it resumes computation from that state. It is convenient to treat each archived component as being unique; for instance, there may be a solid-mechanics computation component that is persistent (and, for practical purposes, lives forever), but an experimenter may have a sequence of related components corresponding to states of that component used at different times in different experiments. Our intent is to provide access to these archived components through a Web browser, using the standard summoning mechanism.

3.6. Related Work

Frameworks are reusable designs for software system processes, described by a set of objects and how those objects can be used (Roberts and Johnson, 1996). Our framework consists of some middleware APIs, a model for using them, and services and patterns that are helpful not only in inheriting from objects, but extending them as well. These features allow the reuse of both design and code, reducing the effort required to develop an application. In this sense, our framework is comparable to other metacomputing, component, and communication frameworks.

3.6.1. Metacomputing Frameworks. Our framework efforts are similar to recent metacomputing endeavors in that we use the Internet as a resource for concurrent computations. *Globus* provides an infrastructure to create networked virtual supercomputers for running applications (Foster and Kesselman, 1996). Similarly, *NPAC at Syracuse* seeks to perform High Performance Computing and Communications (HPCC) activities using a Web-enabled concurrent virtual machine (Fox and Furmanski, 1996). *Legion* is a C++-based architecture and object model for providing the illusion of a single virtual machine to users for wide-area parallel processing (Grimshaw et al., 1996). *Javelin* is a Java-based architecture for writing parallel programs, implemented over Internet hosts, clients, and brokers (Capello et al., 1997). Although our framework could be used for metacomputing applications, we provide neither seamless parallelism nor facilities for developing high-performance applications. Rather, we provide mechanisms for programmers to develop distributed system components and personal networks quickly, and we plan to provide mechanisms for non-programmers to easily customize their components and personal networks.

3.6.2. Component Frameworks. Many other framework systems also have the goal of creating distributed system components. *CORBA* is an architecture for distributed object computing that allows for language-independent use of components through a standardized Object Request Broker (OMG, 1995). *Hector* is a Python-based distributed object framework that provides a communications transparency layer enabling negotiation of communication protocol qualities, comprehensive support services for application objects, and a four-tiered architecture for interaction (Arnold et al., 1996). *OpenDoc* is a component software architecture that allows for the creation of compound documents (MacBride and Susser, 1996).

JavaBeans is a platform-neutral API and architecture for the creation and use of Java components (Java Beans, 1997). *Aglets* provide a Java-based framework for secure Internet agents that are mobile, moving state along with the program components themselves (Lange and Oshima, 1997). We differ from these efforts because our emphasis is not on the implementation of the infrastructure itself; rather, it is on reasoning about global compositional distributed systems with opaque components that have dynamic interfaces and interact by using asynchronous messages.

3.6.3. Communication Frameworks. The *Communicating Sequential Processes* (CSP) model keeps each process active for the entire duration of the computation (Hoare, 1978). As with the language Fortran M (Foster and Chandy, 1995), we implement this model, adding such implementation artifacts as dealing with process setup and removal, and permitting prioritized waits to resolve resource contention. Unlike Fortran M, sessions provide a hybrid technique for running communicating distributed processes which are frozen when they are not performing any work, yet have persistent state that can be revived whenever a new session is initiated.

3.6.4. Collaborative Technologies. Many software products allow collaboration using the Internet. Synchronous collaboration includes *teleconferencing*, provided by applications such as Netscape CoolTalk, Internet Relay Chat, Internet Phone, and White Pine Software CU-SeeMe, and *shared whiteboards*, provided in applications such as CU-SeeMe, wb (Floyd et al., 1995), and Microsoft NetMeeting. State of the art research in agreement protocols (Shenker et al., 1994) has made synchronous collaborations more flexible, but much research remains to be done in infrastructure for asynchronous tools such as concurrent version-control.

4. Summary

This paper describes an ongoing project to archive distributed computations and link these archives into the Web. Our current release of the Infospheres Infrastructure (1.0 beta 2) has mechanisms for saving the persistent state of components. This version of our prototype does not support saving multiple versions or global snapshots, but we have a design in place for this functionality that requires only modest extensions to our current package.

Our preliminary work on a Web of archived distributed computations suggests that the technology has potential benefits. However, several areas remain for future exploration: reducing the storage required for archives by using file differencing and compression, taking partial snapshots of systems when an entire global snapshot is not required, taking snapshots of different parts of the system at varying intervals, and providing support for the deterministic replay of events within a component.

Acknowledgments

Please see `http://www.infospheres.caltech.edu/` for more information about the Infospheres Infrastructure. The idea of archiving computational experiments in asynchronous collaborations and linking the archived experiments was originally suggested by John Reynders and Peter Beckman of Los Alamos National Laboratories.

References

Arnold, D., Bond, A., Chilvers, M., and Taylor, R. (1996). Hector: distributed objects in Python. In conference proceedings – the Fourth International Python Conference (Livermore, June).

Cappello, P., Christiansen, B., Ionescu, M.F., Neary, M.O., Schauser, K.E., and Wu, D. (1997). Javelin: Internet-based parallel computing using Java. Submitted – the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

Chandy, K.M., and Lamport, L. (1985). Distributed snapshots: determining the global states of distributed systems. *ACM Transactions on Computing Systems*, volume 3, number 1, pages 63–75.

Chandy, K.M., and Rifkin, A. (1997). Systematic composition of objects in distributed Internet applications: processes and sessions. In conference proceedings – the Thirtieth Hawaii International Conference on System Sciences (Maui, January), volume 1, pages 395–404.

Chandy, K.M., Kiniry, J., Rifkin, A., Zimmerman, D., Tanaka, W., and Weisman, L. (1997). A framework for structured distributed object computing. Submitted – *Communications of the ACM*, volume 40, number 8.

Chandy, K.M., Rifkin, A., Sivilotti, P.A.G., Mandelson, J., Richardson, M., Tanaka, W., and Weisman, L. (1996). A worldwide distributed sytem using Java and the Internet. In conference proceedings – the Fifth IEEE International Symposium on High Performance Distributed Computing (Syracuse, August), pages 11–18.

Chappell, D. (1996). *Understanding ActiveX and OLE*, Microsoft Press.

Floyd, S., Jacobson, V., Liu, C., McCanne, S., and Zhang, L. (1995). A reliable multicast framework for light-weight sessions and application level framing. In conference proceedings – ACM SIGCOMM (August), pages 342–356.

Foster, I.T., and Chandy, K.M. (1995). Fortran M: a language for modular parallel programming. *Journal of Parallel and Distributed Computing*, volume 26, number 1, pages 24–35.

Foster, I.T., and Kesselman, C. (1996). Globus: a metacomputing infrastructure toolkit. In conference proceedings – the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM (Lyon, August).

Fox, G., and Furmanski, W. (1996). Towards web/Java based high performance distributed computing – an evolving virtual machine. In conference proceedings – the Fifth IEEE International Symposium on High Performance Distributed Computing (Syracuse, August), pages 308–317.

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

Grimshaw, A.S., Wulf, W.A., and the Legion team, The legion vision of a worldwide virtual computer. *Communications of the ACM*, volume 40, number 1, pages 39–45.

Hoare, C.A.R. (1978). Communicating sequential processes. *Communications of the ACM*, volume 21, number 8, pages 666–677.

Infospheres Research Group, The infospheres infrastructure user's guide. Technical report, California Institute of Technology, 1997.

JavaSoft JavaBeans Team. (1997). *JavaBeans*, Sun Microsystems.

JavaSoft Java RMI Team. (1997). *Java RMI*, Sun Microsystems.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, volume 21, number 7, pages 558–565.

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

MacBride, A., and Susser, J. (1996). *Byte Guide to OpenDoc*, Osborne McGraw-Hill.

Mellor-Crummey, J. (1992). Compile-time support for efficient data race detection in shared-memory parallel programs. Technical report – Center for Research on Parallel Computation CRPC-TR92232, Rice University.

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

Peterson, L.L., and Davie, B.S. (1996). *Computer Networks: A Systems Approach*, Morgan Kaufmann.

Roberts, D., and Johnson, R. (1996). Evolving frameworks: a pattern language for developing object-oriented frameworks. In conference proceedings – Pattern Languages of Programs (Allerton Park, September).

Sessions, R. (1996). *Object Persistence Beyond Object-Oriented Databases*, Prentice Hall.

Shenker, S., Weinrib, A., and Schooler, E. (1994). Managing shared ephermeral teleconferencing state: policy and mechanism. In conference proceedings – the International Workshop on Multimedia Transport and Teleservices, (Vienna, November).

Journal of Supercomputing, 11, 19–18 (1997)

Contributing Authors

Mani Chandy received his Ph.D. in Electrical Engineering from MIT. He has been a Professor of Computer Science at Caltech since 1989. He was on the faculty at the University of Texas from 1970 to 1989, and served as Chair. He won the A.A. Michelson award for his contributions to computer performance modeling in 1985. He was a Sherman Fairchild Fellow, and he is an IEEE Fellow and a member of the National Academy of Engineering. He was awarded the 1996 IEEE Koji Kobayashi Award for Computing and Communications. Dr. Chandy has written several books on computer performance modeling and concurrent computing.

Joseph Kiniry received a masters degree in computer science from the University of Massachusetts in 1992 and two bachelors degrees from Florida State University in 1994. He is currently pursuing a Ph.D. in computer science at Caltech, where he works in formal methods, practical models, and interoperability issues for distributed, object-oriented, and multithreaded applications. Prior to attending Caltech, he was a senior researcher at the Open Software Foundation Research Group, where he contributed to several research groups and helped jumpstart the OSF's efforts in Java. For nine years, he has been an independent consultant in distributed systems, software engineering, Internet security and computer graphics.

Adam Rifkin received a masters degree in computer science from the College of William and Mary in 1992, where he worked in parallel processing. He is currently pursuing a Ph.D. in computer science at Caltech, where he works in compositional distributed systems. His efforts with infospheres have won best paper awards both at the Fifth IEEE International Symposium on High Performance Distributed Computing in August 1996, and at the Thirtieth Hawaii International Conference on System Sciences in January 1997. He has consulted and done research with several organizations, including Canon, Hewlett Packard, FoRK Consulting, Roadrunner Technology, Griffiss Air Force Base, and the Nasa-Langley Research Center.

Daniel Zimmerman received a bachelors degree in engineering and applied science in 1996 from the California Institute of Technology, where he learned to fight legends and worked in distributed algorithms, object-oriented programming, and asynchronous VLSI design. He is currently pursuing a Ph.D. in computer science at the California Institute of Technology, where he works in formal methods, practical models, and interoperability issues for distributed, object-oriented, and multithreaded applications. His research efforts, both as an undergraduate and as a member of Caltech's Compositional Systems Research Group, have earned him a three-year graduate research fellowship from the National Science Foundation.