A System for Constructing Configurable High-Level Protocols*

Nina T. Bhatti and Richard D. Schlichting
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

New distributed computing applications are driving the development of more specialized protocols, as well as demanding greater control over the communication substrate. Here, a network subsystem that supports modular, fine-grained construction of high-level protocols such as atomic multicast and group RPC is described. The approach is based on extending the standard hierarchical model of the *x*-kernel with composite protocols in which micro-protocol objects are composed within a standard runtime framework. Each micro-protocol realizes a separate semantic property, leading to a highly modular and configurable implementation. In contrast with similar systems, this approach provides finer granularity and more flexible inter-object communication. The design and prototype implementation running on Mach are described. Performance results are also given for a micro-protocol suite implementing variants of group RPC.

1 Introduction

Network protocols that are implemented at high levels of the protocol stack and that provide rich functionality are increasingly being used to simplify certain types of applications. For example, *ordered atomic multicast* provides atomic and consistently ordered message delivery to a group of processes, which can be useful for writing real-time and fault-tolerant distributed applications [6, 12, 27, 35]. Other high-level protocols of this type include group RPC [8, 9, 10], membership [11, 25, 28], distributed transactions [3], and protocols related to multimedia applications [24, 40]. All provide powerful abstractions that simplify the task of writing applications that must handle uncertainties involved with network communication, distributed synchronization, and processor crashes.

Unfortunately, while such high-level protocols are useful, they embed complex functionality and are therefore difficult to design, debug, and modify. One option for addressing this problem is to implement the functionality as a collection of smaller protocol objects (a *protocol suite*) and then use a system like ADAPTIVE [37], Horus [38], or the *x*-kernel [23] to combine the objects into a network subsystem. Such systems allow the overall functionality to be separated into more manageable modules, thereby accruing advantages in the areas of incremental development, system customization, and code reuse.

Despite their advantages over monolithic realizations, current systems still have a number of deficiencies when it comes to implementing high-level protocols. These include inadequate support for fine-grained modules with complex interaction patterns, limited facilities for data sharing, and an orientation towards hierarchical protocol composition at the expense of more flexible combinations. Experience suggests that these limitations increase the difficulty of implementing high-level protocols using these systems. For example, problems of this type have been encountered with the *x*-kernel, both in Consul, a protocol suite implementing atomic multicast [29, 35], and xAMP, a real-time atomic multicast protocol [39].

In this paper, we describe a new *x*-kernel-based structuring approach that addresses these problems. With our approach, a high-level network protocol is constructed from a collection of *micro-protocol objects* (or just *micro-protocols*) that implement individual semantic properties of the target system.[1] For example, with atomic multicast, one micro-protocol might implement the consistent ordering requirements, while another might implement reliable

*This work supported in part by the Office of Naval Research under grant N00014-91-J-1015.

[1] Our use of the term micro-protocol should not be confused with the *x*-kernel micro-protocols described in [33]. The differences are explained more fully in section 5.

transmission. Micro-protocols can also be used to implement different semantic variants of the same property. For example, with RPC, there may be multiple micro-protocols implementing different policies for how the request is handled if the server fails, such as *exactly once*, *at least once*, or *at most once* semantics [34]. A system is then configured based on the particular properties needed for the given application.

This micro-protocol approach is realized by augmenting the *x*-kernel's standard hierarchical object composition model with the ability to internally structure protocol objects. The result is a two-level model in which selected micro-protocols are first combined with a standard runtime system or *framework* to form a *composite protocol*. This composite protocol, whose external interface is indistinguishable from a standard *x*-kernel protocol, is then composed with other *x*-kernel protocols in the normal hierarchical way to realize the overall functionality required of the network subsystem. Internally, the framework implements an event-driven execution paradigm, in which micro-protocols are executed whenever events for which they are registered—for example, message arrival or a timeout—occur. Thus, when compared with standard *x*-kernel protocol objects, micro-protocols are typically finer-grain objects that interact more closely and do so using mechanisms provided by the framework rather than the *x*-kernel Uniform Protocol Interface (UPI).

Our approach has a number of benefits. For example, the flexibility inherent in the two-level aspect of this model is useful for dealing with dependencies among the constituent properties implemented by these complex protocols. It also offers the development benefits associated with modular implementations, as well as an enhanced ability to tailor the system to the specific characteristics of a given application or architecture. Among other things, this configurability makes the approach suitable for constructing *adaptive systems*, which alter their behavior based on changes in the environment [5, 14]. Our approach is also related to recent work in configurable operating systems [4, 18, 31]. In contrast with similar systems for constructing configurable protocols, our approach provides finer granularity and more flexible inter-object communication, which is especially useful for configuring closely-related service variants of the same general type of high-level protocol (e.g., variants of atomic multicast).

Here, our focus is on describing the design and performance of the prototype implementation, which is integrated with *x*-kernel version 3.2 running on Mach. Below, we first give further motivation and goals, followed by a summary of the services provided by our system and a description of the model. The prototype implementation is then described, together with performance figures from a micro-protocol suite capable of supporting multiple variants of group RPC. Finally, related work is described and conclusions offered.

2 Motivation and Goals

Early protocol systems were designed as monolithic entities, and their implementations reflected this. Even as the layered model gained acceptance as a conceptual tool to view protocol composition, implementations still tended to be ad hoc, reflecting a concern that implementing each protocol as a distinct entity would result in significant performance penalties. It is only recently, in fact, that software support for protocol composition has reached a level where hierarchical collections of protocol objects can be combined into a system whose performance is competitive with monolithic implementations.

Constructing a network service from collections of protocol objects has a number of advantages. Perhaps the most important is that it allows, at least in theory, reuse to construct new services. In other words, a new service can be constructed by writing a new object that implements just the new aspect of the service, and then combining it with existing, well-tested objects that provide the other necessary functionality. Over time, a comprehensive library of objects can be developed, thereby simplifying the development effort, facilitating performance comparisons between protocol implementations, and allowing experimentation with new protocol concepts.

While this hierarchical approach has worked well for a large class of protocols, a persuasive case can be made that it lacks the flexibility needed to implement certain types of protocols. For example, in designing and implementing Consul using the *x*-kernel, a number of inherent problems with the model were discovered [30]. These include the following:

- Provisions for communicating between protocol objects on the same machine are insufficient to implement the necessary complex interactions. In the *x*-kernel, the specific problem is that the UPI lacks sufficient flexibility, thus requiring the programmer to use control operations as a workaround.

- Lack of communication support leads to implicit dependencies between objects, where one object "expects" another to realize some functionality. When compared to an explicit dependency caused by an invocation, implicit dependencies make the software difficult to debug and modify.

- Multiple protocol objects may need to coordinate their actions or synchronize relative to a given message or set of messages. Such coordination is difficult in the current model.

A remarkably similar experience has been reported independently by the developers of xAMP [13].

While these limitations are directly relevant only to atomic multicast protocols, there are several reasons to believe the lessons are applicable to other types of protocols as well. First, increasingly sophisticated services are being implemented as network protocols, in part because of the advent of protocol-oriented kernels such as the *x*-kernel. These services, like atomic multicast, are the type most likely to stretch or break the current model. Second, as distributed applications become more common, the demand for new types of specialized protocols very different from current protocols will increase. Doing such specialization in a hierarchical model—especially fine-grained specialization—is likely to be difficult. Finally, applications are demanding more control over their execution environment, including the communication substrate, in order to achieve the best possible performance. Such configurability will further increase the complexity and variety of protocols that must be supported.

This research is based on the premise that the construction of network services through the composition of protocol objects is the appropriate paradigm. Our objective, however, is to relax the restrictions on communication among objects on a single host imposed by the hierarchical approach. In our approach, protocol objects performing unrelated tasks are located in adjacent layers and communicate normally using the standard UPI of the *x*-kernel. However, protocol objects that need to communicate more often or cooperate more fully—our micro-protocols—are co-located within a structure that provides richer facilities for this type of interaction. Micro-protocols have no direct knowledge of each other; communication is achieved indirectly through an event mechanism.

This structure, described in detail in the next section, has a number of benefits, including:

- *Expressibility*. The micro-protocol execution environment provides a new, more general model for structuring protocol objects. Micro-protocols can communicate with an arbitrary number of other micro-protocols, can synchronize when necessary, and can operate on collections of messages. The environment also supports multiple threads of execution.

- *Configurability*. A network service is constructed out of modular micro-protocols, each of which implements a specific semantic property. The result is an approach that supports a high degree of configurability and the construction of services that are customized to the needs of the application.

- *Efficiency*. Since a network service can be customized, the application avoids execution overhead that can result from the inclusion of unnecessary properties. For example, it is easy to build an atomic multicast that includes no consistent ordering of messages, thereby avoiding the delay inherent in doing such ordering.

- *Reusability*. Micro-protocols implementing various semantics can be used in multiple services. For example, a liveness micro-protocol that checks that all processes have sent a message within some given time interval can be used in a variety of protocol suites.

- *Ease of debugging and maintenance*. Since a service is constructed from small micro-protocols, each can be debugged and maintained independently. Although the compatibility of combinations of micro-protocols must still be verified, this process is simplified since interactions between micro-protocols are largely explicit.

- *Explicit dependencies*. Dependencies between micro-protocols are explicit since "back door" communication channels are unnecessary. This makes understanding the micro-protocols easier and the interactions obvious.

- *Future opportunities for optimization*. Explicit dependencies create the potential for code optimization. For example, it may be possible to in-line code using techniques similar to [1] to yield a system with efficiency competitive to monolithic implementations.

- *Availability of x-kernel protocols*. Since our system is incorporated in the *x*-kernel, all existing and future *x*-kernel protocols can be used without modification.

In summary, then, our goal is to extend current technology to encompass more fine-grained composition of protocol objects, both to simplify development and to increase the configurability of the network subsystem.

3 Constructing Composite Protocols

In the standard *x*-kernel model, a hierarchical graph of protocol objects is used to realize a communication service. A thread shepherds each message along a path through the graph executing the *x*-kernel operations call, push, pop, and demux to route the message on the correct path from the application to the network or vice versa. Messages can be modified, destroyed or created as they traverse the graph.

In addition to processing application messages, protocol objects can use messages to communicate with other protocol objects to which they are connected in the graph. Since this graph is hierarchical, however, communication flexibility is limited, especially with regard to allowing communication among protocol objects at the same level of the graph. Thus, our scheme augments this model by adding composite protocols, which essentially create new

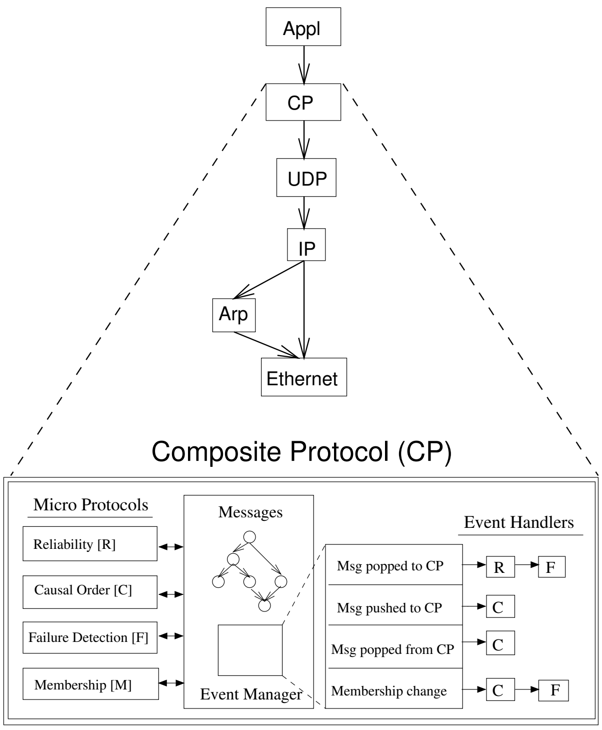

Figure 1: Composite protocol within an *x*-kernel protocol graph.

ways for protocol objects at the same level to communicate. In addition, we have extended the one-thread-per-message model to multiple-threads-per-message model and provided an event-driven mechanism for protocol communication.

3.1 A Two-Level Model of Protocol Composition

In our model, the standard *x*-kernel hierarchical model is augmented with the ability to include composite protocols in the protocol graph in conjunction with simple *x*-kernel protocols. Unlike simple protocols, each composite protocol has an internal structure formed of a collection of micro-protocols executed in an event-driven manner. The major components of a composite protocol are:

- *Micro-protocols*: A section of code that implements a single well-defined property or provides some specific functionality. Consists of header information, private data, initialization code, and a collection of event handlers. May export data for use by other micro-protocols.

- *Events*: An occurrence that causes one or more micro-protocols to be invoked. Event handlers are invoked (logically) in parallel. Event types specify whether the triggering micro-protocol is blocked until completion or not. Some events of interest are predefined (e.g., message arrival); others are defined by micro-protocols (e.g., change in group membership).

- *Framework*: A runtime system that implements the event registration and triggering mechanism, and contains shared data (e.g., messages) that can be accessed by more than one micro-protocol.

micro-protocol *name* {
  ... *Decl of exported events, message attributes,*
      data inspection, modification routines ...
  ... *Decl of imported events, global variables ...*
  ... *Decl of private events, message attributes, variables ...*
  ... *Initialization code ...*
  ... *Event handlers ...*
  ... *Data inspection routines ...*
  ... *Local procedures ...*
} end micro-protocol *name*

Figure 2: Micro-protocol schema

An example of this model is shown in Figure 1. Above is an *x*-kernel protocol graph that contains a composite protocol CP implementing atomic multicast. Below is an expanded view of CP illustrating the components of the model. In the middle of CP is the runtime framework, which contains a shared data structure—in this case a bag of messages—and some event definitions. The boxes to the left represent micro-protocols, while to the right are some common events with the list of micro-protocols that are to be invoked when the event occurs.

3.2 Micro-Protocols

A micro-protocol is structured as shown in Figure 2. Events relevant to a micro-protocol are declared as either *exported* or *imported*. An exported event is one that is raised by the micro-protocol, while an imported event is one for which the micro-protocol provides a handler. The micro-protocol may also contain private events that are used for internal communication and declare private data visible to all handlers defined in the micro-protocol.

Data inspection routines are exported when a micro-protocol maintains information and wants to make it available to other micro-protocols. For example, a membership micro-protocol might export a routine that returns the current membership list. In these situations, only the micro-protocol declaring the data can alter it, so that changes by other micro-protocols must be requested by raising an event or calling an exported routine that modifies the data. When a micro-protocol modifies its data, it will often raise an event to notify other micro-protocols about the state change. For example, the membership micro-protocol might react to a "timeout" event by suspecting that a process has failed. If, after further checking—for example, by running an agreement protocol with the other processes—it determines that a failure has indeed occurred, it would update the membership list and raise an event declaring a change to that list.

We express micro-protocols in an informal protocol description language that supports the structure of micro-protocol programming described above, and enforces visibility and modularity rules. An example micro-protocol written in this pseudo-code is given in section 3.5.

Other aspects of micro-protocols shown in Figure 2 (e.g., message attributes) are described below.

3.3 Events and Handler Execution

Events are a general communication mechanism used to inform micro-protocols that something of interest has happened. A micro-protocol requests notification from the runtime system for a given event by declaring a handler as shown above. Each event may have multiple handlers, which can be useful when multiple micro-protocols need to be notified that the event has occurred. For example, the arrival of a message from the network may trigger one handler involved with detecting host failures and another that is involved with inserting the message into an ordering graph.

Handlers are not necessarily known to the micro-protocol raising the event. This property helps decouple micro-protocols from one another, thereby simplifying the task of writing micro-protocols that can be combined in a flexible fashion with other micro-protocols. As an example of this decoupling, one micro-protocol can be responsible for detecting a situation, with another implementing the policy for resolving it. This type of structure allows the policies for each to be realized orthogonally based on the needs of the application and the specific collection of micro-protocols configured into the framework.

Events can also have parameters. For example, when an event corresponding to the expiration of an acknowledgment timer occurs, we might also want to communicate which message is lacking the acknowledgment. Such functionality can be realized by passing that information as an argument to the registered event handlers. All parameters are passed by value.

Events can either be user-defined or predefined by the runtime system. A user-defined event, such as the one related to timer expiration above, is exported (declared) by a given micro-protocol and explicitly raised by invoking a routine implemented by the framework. Predefined events, on the other hand, are exported by the runtime framework and implicitly raised when the framework detects that the event has occurred. In both cases, the event can be imported (handled) by any number of other micro-protocols.

The following list gives the predefined events currently supported; here, xMsg refers to an *x*-kernel message and CPMsg refers to a composite protocol message, both of which are described in more detail in section 4.1 below:

- Message_Popped_To_CP(xMsg): An *x*-kernel message from a lower level *x*-kernel protocol has been popped to the composite protocol.
- Message_Popped_From_CP(CPMsg): A message has been popped from the composite protocol to the *x*-kernel higher level protocol.
- Message_Pushed_To_CP(xMsg): An *x*-kernel message from a higher level *x*-kernel protocol has been pushed to the composite protocol.
- Message_Pushed_From_CP(CPMsg): A message has been pushed from the composite protocol to the *x*-kernel lower level protocol.
- Message_Inserted_Into_Bag(CPMsg): A message has been constructed and inserted into the shared bag of messages.
- Message_Deleted_From_Bag(CPMsg): A message has been deleted from the shared bag of messages.
- Message_Ready_To_Be_Sent(CPMsg): All micro-protocols are satisfied that the message can leave the composite protocol, either to be popped or pushed.

In addition, there are provisions for timer events that are generated after a specified amount of time has passed.

A micro-protocol can only handle events that it declares as imported, or private events that are locally generated and handled by the same micro-protocol. An event must be exported by a micro-protocol to be imported by another micro-protocol.

Handlers are scheduled for execution when an event is raised. If there are multiple handlers registered for that event, the order in which they are executed is indeterminate. In fact, they may be executed in parallel given the appropriate hardware.

Execution of a micro-protocol that raises an event can either block until all handlers have completed (*synchronous*) or proceed without blocking (*asynchronous*). The choice of semantics is specified as an argument in the system call that raises an event, implying that it can vary on a per-invocation basis. These semantics extend as expected through multiple levels of recursively raised events.

3.4 Framework

The framework is a runtime system that implements the event mechanism and provides a shared bag of messages on which micro-protocols operate. It also implements an *x*-kernel compliant interface for the composite protocol, which enables it to interoperate with other *x*-kernel protocols in the standard way.

The framework accepts messages from the *x*-kernel and transfers control to micro-protocols by raising the appropriate events and executing the appropriate event handlers. As already noted, in the *x*-kernel, one thread shepherds any given message through the entire protocol graph, executing code in various protocol objects on behalf of the message. To handle the execution of potentially many event handlers, however, we extend this model to allow multiple threads to execute on behalf of the message during its residence in the composite protocol.[2] This model provides more flexibility than the one thread per message in the context of composite protocols, and also allows the possibility of true parallel execution, as noted above. The one thread per message model is restored when a message leaves a composite protocol and is handed over to a standard *x*-kernel protocol object.

Messages that arrive at a composite protocol are placed in an unordered bag of messages maintained by the framework that functions as a global pool accessible to all micro-protocols. This feature is intended to support two aspects of programming that are common in the type of high-level protocols for which this approach is intended. First, it allows micro-protocols to make state changes based on information in an entire collection of messages, rather than just a single message. This can be important, for example, in an atomic multicast protocol that requires waiting for a collection of messages to arrive and then deterministically sorting the collection before presenting messages to higher levels [29, 35]. Second, a shared bag of messages allows multiple micro-protocols to access messages concurrently. This can be important, for example, in a situation where a message is acknowledged by one micro-protocol while concurrently being ordered relative to other messages by a second micro-protocol.

Prior to being placed in the bag, a verify micro-protocol is executed to determine if the message is acceptable. For instance, a message might be rejected if corruption is detected or if it is destined for a process that no longer exists. If the message is acceptable, the verifying micro-protocol places the message in the bag using a routine provided by the framework. A default version of the micro-protocol is provided with the framework, although an alternative can easily be substituted by the user to perform message screening and bag insertion under program control.

Each message in the bag has a collection of *attributes* that encode certain types of per-message information. *Predefined attributes* are supplied by the framework. For example, one such attribute is direction, which indicates whether the message is being sent up or down the *x*-kernel graph. *Micro-protocol attributes* contain micro-protocol-specific information about the message. For example, a reliability protocol may keep private state information about the message indicating whether it was acknowledged or is being retransmitted and by which hosts. Such attributes can be declared either private or public; a private attribute is visible only to the micro-protocol that defines it, while a public attribute can be read by all micro-protocols. In addition, attributes are used to build headers for messages that are pushed from the framework. This is done by an attribute-to-header routine provided by the user and invoked by the framework as a message is exiting the composite protocol. Similarly, when a message is popped to the framework, a header-to-attribute mapping routine is invoked that unpacks the header and creates attributes using this information. As with verify, default mapping routines are supplied, but can be overridden by the user if desired.

As already noted, data defined within a micro-protocol can also be shared by exporting appropriate inspection routines. Any necessary synchronization within these routines is done explicitly using semaphores. With our prototype implementation, such synchronization is only necessary if the data is not written atomically and either a message push or an explicit event triggering is done in the middle of the code effecting the change.

[2] This is optimized in the implementation to a series of procedure calls on sequential hardware; see section 4.

3.5 Example: Micro-Protocols from a Group RPC Suite

To illustrate the structure of micro-protocols and the event-driven programming paradigm, we present two short examples of micro-protocols that might be part of a suite used to implement a group RPC service. The first is a simple membership micro-protocol that updates a membership list whenever a host is suspected of having failed. Note that the indication of this suspicion would most likely be in the form of an event generated by another micro-protocol, such as a liveness micro-protocol that tracks message arrivals. The second is an acknowledgment micro-protocol that sends an ACK message for each reply message received, sends a "still working" message to a client if the reply from the local server is slow, and raises the Suspect_Host_Dead event if a server is suspected to have failed.

Membership Micro-Protocol. Figure 3 shows the code for the membership micro-protocol. At the top is an exports section that specifies inspection routines, events, and attributes that are exported for use by other micro-protocols. Here, an event for membership change and a routine for ac-

```
micro-protocol MEMBERSHIP {
    exports{
        event  Membership_Change(ch_t type);
        proc   memList_t GetGroup();
    }
    imports{
        event  Suspect_Host_Dead(mem_t host);
    }
    private{
        memList_t     MemberList;
    }
    initialize{
        initMembershipList();
    }
    actions{
        Suspect_Host_Dead(mem_t host)   →
            if (find(host, MemberList)) {
                deleteMember(MemberList, host)
                raiseEvent(Membership_Change,
                    DELETION, ASYNC)
            }
    }
    ... code for deleteMember, GetGroup, and
            initMembershipList ...
} end micro-protocol MEMBERSHIP
```

Figure 3: Simple membership micro-protocol

cessing the current group membership are provided. Note that the event specification includes a parameter to indicate whether the event of interest is the failure or recovery of a host. The exports are followed by an imports section, in this case an event corresponding to a suspected failure. This particular event is raised by the acknowledgment micro-protocol below and fielded by an event handler in MEMBERSHIP. Note that this specification also includes a parameter, specifically, an indication of which host is suspected to have failed. Next, the micro-protocol includes declarations for any private data, attributes, and events. In this case, the only private data is the membership list maintained by the micro-protocol.

The declarations are followed by the procedures that make up the body of the micro-protocol. The first is an initialization routine, which initializes the membership list from some external source; for example, it may be read from a file. This routine is executed, in *x*-kernel terms, at initialization time prior to execution of the standard open or openenable routines.

After the initialization code is the actions section, which contains the event-handling code. The general form of an action is:

event name [&& boolean-expr]* $\rightarrow$ handler

The optional boolean expression is used to make the handler execution conditional. The expression may reference event parameters, message attributes, and micro-protocol variables. In MEMBERSHIP, there is one handler that deletes a member from the list when the Suspect_Host_Dead event is triggered. The parameters to the event are available to the handler, as is any private data declared within the micro-protocol.

The remainder of the micro-protocol contains inspection routines for export, local procedures, etc. In this micro-protocol, there are three such routines: deleteMember, GetGroup, and initMembershipList. Their code is omitted here for simplicity.

Acknowledgment Micro-Protocol. Figure 4 shows the code for a simple acknowledgment micro-protocol ACK that generates the Suspect_Host_Dead event when a message has not been acknowledged after some interval of time. This interval can be adjusted by a call to the setInterval routine. The timer is started at the time the message is pushed from the composite protocol (note the import of the Message_Pushed_From_CP event). The timer is set by the setTimerEvent call, which gives the interval to wait and an indication that this event is to be generated only once rather than periodically. This Timeout event is declared in the private section of the protocol and is therefore raised and handled only by ACK.

The second set of tasks done by ACK involve acknowledging any messages that are received. It accomplishes this by handling the Message_Inserted_Into_Bag event for messages of type REPLY. The event is qualified so that only reply messages are acknowledged. Request messages are only acknowledged if the server is slow in responding, which is also handled using the Timeout event. The server and client sides of the communication are handled by the same micro-protocol, with the imported state variables server and client being used in the code to distinguish between the two.

4 Implementation and Performance

Our prototype implementation is based on *x*-kernel version 3.2 running on Mach version MK82 and runs as a user-level task. The prototype is written in C and is structurally a collection of library routines that is linked with the user-written micro-protocols to generate a composite protocol. After such a protocol has been produced, it is included in the *x*-kernel protocol graph in the normal way.

Here, we focus on describing the implementation details of the runtime framework since much of the system's functionality is implemented there. Initial performance results from a group RPC micro-protocol suite and a null protocol test are also given.

```
micro-protocol ACK {
    exports {
        event  Suspect_Host_Dead(mem_t host);
        proc   SetInterval(int millisec);
    }
    imports {
        event  Message_Pushed_From_CP(CP_Msg_t msg);
        event  Message_Inserted_Into_Bag(CP_Msg_t msg);
        boolean   client, server;
    }
    private  {
        event       Timeout(CP_Msg_t msg);
        attribute   serverList_t servers ;
        int         interval;
    }
    initialize{
        InitTimerVal();
    }
    actions {
        /* Set timer event to detect message loss*/
        Message_Pushed_From_CP(CP_Msg_t msg) &&
            client && msg.attr.type == REQUEST  →
            setTimerEvent(Timeout, CP_msg, interval,
                ONCE);

        /* Set timer event to ensure timely reply */
        Message_Inserted_Into_Bag(CP_Msg_t msg) &&
            server && msg.attr.type == REQUEST  →
            setTimerEvent(Timeout, CP_msg, interval,
                ONCE);

        /* Send ACK message */
        Message_Inserted_Into_Bag(CP_Msg_t msg) &&
            client && msg.attr.type == REPLY  →
            sendAckToSender(msg,REPLY_RECEIVED);

        /* Send "still working" message if server slow.*/
        Timeout(CP_Msg_t msg) && Server  →
            sendAckToSender(msg, STILL_WORKING );

        /* Suspect server failure */
        Timeout(CP_Msg_t msg, host) && Client  →
            if ( hostNotResponding(msg, host)) {
                RaiseEvent(Suspect_Host_Dead,
                    host, ASYNC);
            }
    }

    ... code for SetInterval, InitTimerVal, sendAckToSender,
                and hostNotResponding ...

} end micro-protocol ACK
```

Figure 4: Simple acknowledgment micro-protocol

4.1 Framework

Uniform Interfaces. The framework encapsulates the micro-protocols and delivers messages to and from other *x*-kernel protocols. To accomplish this, the framework provides the standard *x*-kernel interface operations, such as call, push, pop, and demux. These allow composite protocols to be added to an existing *x*-kernel protocol graph without requiring changes to the existing protocols. The framework can be configured to provide a synchronous call interface or an asynchronous push interface, to accommodate both styles of *x*-kernel protocols. A call-style protocol is blocked when doing a call operation and is unblocked only after the reply message has been filled in. If the push style is used, the caller is not blocked and the reply message (if any) is returned asynchronously.

Thread Management. As noted in the previous section, multiple threads of control may be spawned in the course of executing event handlers. In the prototype, the *x*-kernel thread facility based on Mach C-threads is used as the underlying mechanism. The choice to use this facility rather than spawn C-threads directly was made primarily for two reasons. One is that this makes the threads visible to the *x*-kernel, and, in particular, its built-in features for doing execution monitoring and debugging. This allows the programmer to exploit these features, thereby simplifying the programming process. The other reason is that it allows us to exploit the *x*-kernel's optimized thread management. In particular, since the *x*-kernel preallocates a pool of C-threads at initialization time and manages them directly, the cost of performing thread creation at runtime is avoided.

Thread management is simplified even further in a second version of the prototype in which handler invocations are implemented by procedure calls rather than by explicitly forking a thread. This optimization is targeted for sequential machines where a procedure call is typically more efficient than spawning a thread. No changes are required in the code for the micro-protocols. In fact, which version of the runtime is used is transparent to both the *x*-kernel and the protocol writer.

We also alter the *x*-kernel thread structure by assuming control over a thread that enters the composite protocol. In general, it will execute some sequence of event handlers and then a push or pop to exit the composite protocol. Alternatively, it can simply terminate within the protocol after the last event has been handled. The thread behavior is naturally different depending on whether handler execution is implemented by threads or procedure calls. In the thread implementation, the thread that enters the composite protocol returns to the caller after raising the first event. Once the event is raised, other threads are activated to execute the handlers. On the other hand, with the procedure-based implementation, the entering thread executes each event handler until all handlers are executed (recursively) and then returns to the caller. Timing event are necessarily implemented as threads and are based on the *x*-kernel timer events.

Bag of Messages. A CP message is a structure that contains an *x*-kernel message pointer, attributes, and send bits.

The attributes are created by combining the attribute declarations from all micro-protocols into a "super structure" of attributes. There is one send bit for each micro-protocol. When all send bits are set, the CP Message is ready to be sent and Message_Ready_To_Be_Sent event is raised.

The following operations are provided for manipulating the shared bag of messages:

- Item = newItem(xMsg, direction): Allocates and initializes a new bag item; returns a handle to the appropriate structure. direction indicates if the message was traveling up or down through the *x*-kernel protocol graph when it entered the composite protocol.

- insertItem(CPMsg): Inserts CPMsg into the bag. Automatically triggers the Message_Inserted_Into_Bag event.

- deleteItem(CpMsg): Removes CPMsg from the bag, but does not deallocate storage for the item. Deallocation is done under micro-protocol control, although a message is usually deallocated as soon as it is deleted unless needed for retransmissions, etc. Automatically triggers the Message_Deleted_From_Bag event.

- empty(): Removes all messages in the bag.

- n = count(): Returns a count of the number of items in the bag.

- setSendBit(ProtocolID, CPMsg): Sets the send bit for ProtocolID. When all bits are set, the Message_Ready_To_Be_Sent event is triggered.

- sprintItem(string, CPMsg): The current state of CPMsg (including attribute values) is placed into string. Useful for debugging.

- printBag(): Prints the current contents of the bag to stdout. Useful for debugging.

Micro-protocols written in the protocol description language psuedo-code are currently translated by hand into C files that are compiled using the standard C compiler. We enforce visibility rules using C externs and static declarations.

CP Messages. CP messages are based on *x*-kernel messages, which are optimized for message manipulations such as header pushes and pops, fragmentation, and assembly. The usual *x*-kernel message operations are still supported, but we add additional information in the form of attributes that are efficiently accessed. The scope of micro-protocol attribute names is limited to the micro-protocol in which they are declared, but public attributes must have globally unique names.

Implementation Portability. The runtime framework relies only on facilities provided by the *x*-kernel. As a result, it it is automatically portable to another environment that has a working *x*-kernel implementation. No Mach facilities are used directly.

4.2 Performance

We present two sets of initial performance results. The first is based on a group RPC micro-protocol suite that supports multiple variations of the protocol, while the second is from a null composite protocol in which a series of event handlers are executed to test overhead. All measurements were done on DecStation 5000/240s connected by a 10 Mb Ethernet. The network was not completely isolated from other traffic, but was separated from the main departmental network by a bridge. Also, an effort was made to perform the tests during periods of relatively low network activity.

The first set of performance experiments measure several configurations of a group RPC composite protocol Group_RPC. These configurations were constructed from a collection of micro-protocols implementing different properties of such a service:

- BOUNDED (BND): Provides for bounded termination of the client's request, i.e., either the request is executed within some interval or an exception is returned.

- UNBOUNDED: No *a priori* bound is set on a client's request.

- FIFO: Forces FIFO ordering of client requests at a server; if not included, the server may receive requests from a given client in any order.

- MEMBERSHIP (MEM): Maintains a simple membership list of active hosts; hosts are added as a result of request messages to the CP, and deleted if they do not respond within a given time interval.

- ACK: Acknowledges request messages and handles timeouts.

- UNIQUE: Eliminates duplicate request or reply messages.

- ONE_ACCEPT (1ACC): Implements a policy of accepting the first reply from any server as satisfying the client's request.

- ALL_ACCEPT (AAC): Implements a policy of collecting replies from all the servers.

- SYNC: Provides synchronous request/reply call-style interface to CP.

- ASYNC: Provides asynchronous push-style interface to CP.

System Configuration	Servers	avg
x-kernel Sun RPC	one	4.38
GRPC,SYNC,1AC,MEM	one	6.82
(same)	two	8.90
GRPC,SYNC,AAC,MEM	two	8.45
GRPC,ASYNC,FIFO,1AC MEM	one	6.22
GRPC,ASYNC,FIFO,1AC MEM,BND	one	6.45

Table 1: Time for Group_RPC call (in msec)

- GRPC: Verifies incoming messages, initializes state attributes, and maintains client and server state information; required for any combination of micro-protocols.

The tests consisted of a client sending a 4-byte integer to one or more servers, which respond with an integer. Each test makes 1000 RPC calls and was run 10 times. The roundtrip times are the average of the 10 test runs. To provide a baseline, a version of Sun RPC implemented using the standard *x*-kernel was also tested. Note, however, that Sun RPC is a peer-to-peer rather than group protocol, and, as a result, implements less functionality than Group_RPC.

The tested configurations can be summarized as follows:

- GRPC, SYNC, 1AC, MEM: Implements a group RPC service that provides a synchronous call interface and returns when the first response is received (i.e., a one accept policy). One client and one or two servers were used.

- GRPC, SYNC, AAC, MEM: Identical to the above, but with an all accept policy, which causes the client to wait until responses from all servers are received. Test data is given for two servers. Such a configuration might be used, for example, in a simple replicated database, where the application must know that each group member has completed the request before continuing.

- GRPC, ASYNC, FIFO, 1AC, MEM: Implements an asynchronous call with FIFO ordering. The ordering micro-protocol ensures that the server executes all calls from a given client in FIFO order, despite the possible variations introduced by the asynchronous call interface and network transmission.

- GRPC, ASYNC, FIFO, 1AC, MEM, BND: Identical to the above, but with the addition of bounded termination. This micro-protocol ensures that the client call returns within a bounded interval.

The average rountrip times for the various configurations are given in Table 1. The relative ordering is what one would expect: normal Sun RPC using the *x*-kernel is fastest, while the two configurations requiring communication with two servers are the slowest. Overall, we feel that the figures are reasonable, especially given the preliminary nature of the prototype and the level of functionality provided to the application. We also observed a low variance, ranging between 8.7 and 35 microseconds.

As noted, the *x*-kernel Sun RPC is included only for comparison. Such a protocol would naturally be used for simple client/server communication, but does not provide the multiple acceptance policies, group membership, multiple servers, or message ordering options needed for more complex applications.

Additional micro-protocols are currently being implemented, including other reply ordering policies, two different policies for handling orphan computations, and provisions for atomic execution of requests by the server. [21] elaborates further on various abstract properties of RPC and describes the different combinations that are possible using our micro-protocol approach.

The second set of performance figures are from a null composite protocol designed to measure the event mechanism. Each of the tests measured roundtrip message transmission times for two processes using the experimental network described above. The first is a normal *x*-kernel implementation of UDP without composite protocols; this provides a baseline. In the second, a composite protocol using the procedure call event implementation (CP-P) is inserted between the UDP protocol and user program on both the client and server sides. On the client side, CP-P simply passes messages and acknowledgments to the UDP protocol and user program, respectively, with no changes. On the server side, CP-P generates an acknowledgment for each message, as well as passing it through to the user program. 19 events are generated for each message round trip, and 19 handlers are invoked. The third test is identical, except that a runtime framework with the thread-based event mechanism is used. This composite protocol is called CP-T. Figure 5 illustrates the structure and message flow of the second and third configurations.

The results are shown in Table 2. Although these numbers clearly indicate some overhead, the results are encouraging. Based on the one byte test, each event handler activation costs no more than 33.7 microseconds for procedure-based event dispatching and 206 microseconds for thread based. Note that this figure includes amortizing all execution costs associated with a composite protocol over the handler activations, not just the cost of the invocation itself. The variance was again observed to be low.

Although we have not yet profiled the system in detail, our belief is that the vast majority of the time in both sets of experiments is attributable to the overhead imposed by

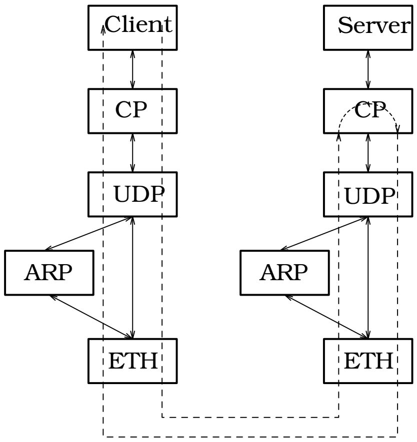

Figure 5: Experimental configuration

Packet Size	*x*-kernel UDP	+CP-P	+CP-T
1 byte	1.57	2.2	5.48
1 K	4.18	4.84	8.19
2 K	7.39	7.89	11.38
4 K	12.65	12.93	16.96
8 K	23.77	23.78	27.63

Table 2: Roundtrip time for null CP (in msec)

executing as a user task on Mach. Our intent is to port the system to the new Scout operating system [31], which should yield more reliable estimates of the overhead of our approach.

5 Related Work

A number of other papers have addressed areas related to this work. Several are in the area of fault-tolerance, where researchers have explored use of modularization or system customization. Examples include the ANSA system [32] and the work on multicast reported in [16]. In contrast to these, our approach is more general and provides more flexibility for the protocol designer. Also in the area of fault-tolerance, [7] explores orthogonal properties of transactions. Such characterizations are complementary to our work since they suggest applications that might be suitable for implementation using our model.

Another area of related work concerns development of system support for constructing modular protocols. The *x*-kernel itself is, of course, one such system. Our work is an extension of the *x*-kernel model, with the goal of supporting finer-grain protocol objects that require richer facilities for communication and data sharing, while retaining the programming and configurability advantages of the *x*-kernel. As noted above, the need for such facilities has been directly motivated by earlier experience using the *x*-kernel to construct the type of high-level protocols that are the target of this research [30]. Many of our goals related to system customization, code reuse, and protocol configurability are adopted from the *x*-kernel.

Other *x*-kernel related work has explored the use of finer-grain protocol objects [33], but the emphasis there is on syntactic decomposition of higher-level protocols within a hierarchical framework. This work, however, does lend credence to the claim that such fine-grain modularity can be introduced without sacrificing performance. System V Streams [36] also supports modularization of protocols, but its model is also hierarchical and relatively coarse-grained. Horus [38] supports stack-line configurations of coarse-grained protocols.

Somewhat closer to our work is the ADAPTIVE system [37], which is also designed to support flexible combinations of protocol objects. The goal of the system is to support efficient construction of transport services with different quality-of-service (QoS) characteristics, especially for multimedia applications using high-performance networks. In contrast with our work, the designers of ADAPTIVE emphasize runtime reconfiguration, automatic generation of *sessions*—i.e., instances of protocol objects—from high-level specifications, and support for alternative process architectures and parallel execution. Moreover, the type of protocol objects supported appear relatively coarse-grained when compared to our objects—multicast rather than individual properties of multicast, for instance—and more oriented toward hierarchical composition and limited data sharing.

Several other efforts have also concentrated on supporting parallel execution of modular protocols, including [15, 26]. While similar to our work in the sense of decomposing protocols along semantic lines, these efforts differ in their emphasis on using parallel execution to improve throughput and latency for high-performance scientific applications. They also retain a single-level composition model, which we believe does not offer enough flexibility for high-level protocols.

Finally, recent work on new generation operating systems has emphasized similar customization goals, but in a more general context [4, 18, 31]. These projects attempt to increase the ability of users to configure different types of services, but for many aspects of operating system functionality rather than just network protocols. However, the configurability they provide is typically more coarse-grained than our approach, which emphasizes choice among specific semantic properties of high-level protocols.

6 Conclusions

High-level protocols such as those found in fault-tolerant distributed systems are becoming increasingly prevalent in a variety of application areas. In addition to being large and difficult to construct, such protocols often have many variants, each of which implements a slightly different semantics. Here, we have described a system for implementing configurable versions of these protocols in which fine-grained micro-protocol objects are composed using a runtime framework to yield an *x*-kernel compatible composite protocol. With this approach, micro-protocols can be written to realize individual semantic properties, with interactions between micro-protocols confined primarily to the raising and handling of events. This facilitates modularization of the software needed to realize each property, while still allowing the flexibility needed to implement the necessary communication and synchronization. Such an approach simplifies the construction of such protocols, as well as allows the construction of customized services with properties tailored to the needs of a given application. It also encourages experimentation with different communication substrates for a given application.

The topic of when and how micro-protocol variants can be configured into a system is addressed in [19, 21, 22] for different types of network services, including membership and group RPC. The use of this approach for constructing a customized atomic multicast protocol for a version of the Linda coordination language with fault-tolerance extensions [2] is described in [17].

The prototype implementation described in this paper illustrates the feasibility of extending the *x*-kernel to support this two-level model of composition. In the prototype, messages arrive at a composite protocol and generate events that result in handlers in the appropriate micro-protocols being invoked to deal with the message. These handlers are executed by separate threads, or, in a version optimized for sequential machines, by a single thread using a series of procedure calls. Initial use of the prototype for implementing different variants of group RPC demonstrates the feasibility of our approach, as well as quantifies the overhead of the event-driven execution model. These results suggest that this approach will not only lead to improvements in the configurability and system customization aspects of high-level protocols, but also yield highly modular implementations that are more than competitive with current monolithic implementations.

Acknowledgements

The composite protocol programming model was developed in conjunction with Matti Hiltunen. Wanda Chiu implemented the group RPC micro-protocol suite and assisted in conducting the performance tests in section 4. The comments from the anonymous referees were very helpful in the revision process, and we especially appreciate the input and guidance of Ellen Siegel.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Trans. on Networking*, 1(5), Oct 1993.

[2] D. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Trans. on Parallel and Distr. Syst.*, 6(3):287–302, March 1995.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[4] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - an extensible microkernel for application-specific operating system services. *ACM Op. Syst. Review*, 29(1):74—77, Jan 1995.

[5] T. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Trans. Comput. Syst.*, 9(2):143–174, May 1991.

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug 1991.

[7] A. Black. Understanding transations in an operating system context. *ACM Op. Syst. Review*, 20(1):73–76, Jan 1991.

[8] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM'86*, pages 406–415, Aug 1986.

[9] R. Cmelik, N. Gehani, and W. D. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings of the 18th IEEE Symp. on Fault-Tolerant Computing*, pages 55–61, Tokyo, June 1988.

[10] E. C. Cooper. Programming language support for multicast communication in distributed systems. In *Proceedings of the 10th IEEE Conf. on Distr. Computing Syst.*, pages 450–457, Paris, France, 1990.

[11] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[12] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th IEEE Symp. on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[13] H. J. F. Fonseca. Support environments for the modularization, implementation and execution of communication protocols. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, June 1994. In Portuguese.

[14] J. Goldberg, I. Greenberg, and T. Lawrence. Adaptive fault tolerance. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 127–132, Princeton, NJ, Oct 1993.

[15] M. Goldberg, G. Neufeld, and M. Ito. The parallel protocol framework. Technical Report 92-16, Dept. of Computer Science, University of British Columbia, Vancouver, British Columbia, Aug 1992.

[16] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Dept of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, 1992.

[17] D. Guedes, D. Bakken, N. Bhatti, M. Hiltunen, and R. D. Schlichting. A customized communication subsystem for FT-Linda. In *Proceedings of the 13th Brazilian Symposium on Computer Networks*, Belo Horizonte, MG, Brazil, May 1995. To appear.

[18] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symp. on Operating System Principles*, pages 69–79, Asheville, NC, Dec 1993.

[19] M. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symp. on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, Oct 1993.

[20] M. Hiltunen and R. D. Schlichting. A model for adaptive fault-tolerant systems. In K. Echtle, D. Hammer, and D. Powell, editors, *Dependable Computing—EDCC-1 (Proceedings of the 1st European Dependable Computing Conference), Lecture Notes in Computer Science, Vol. 852*, pages 3–20. Springer-Verlag, Berlin, 1994.

[21] M. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th IEEE Conf. on Distr. Computing Syst.*, Vancouver, BC, May 1995. To appear.

[22] M. Hiltunen and R. D. Schlichting. Properties of membership services. In *Proceedings of the Second IEEE Symp. on Autonomous Decentralized Systems*, pages 200–207, Phoenix, AZ, April 1995.

[23] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. on Softw. Eng.*, 17(1):64–76, Jan 1991.

[24] R. Keller and W. Effelsberg. MCAM: An application layer protocol for Movie Control, Access, and Management. In *Computer Graphics (Multimedia '93 Proceedings)*, pages 21–30. ACM, Addison-Wesley, Aug. 1993.

[25] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, Wien, 1991.

[26] B. Lindgren, M. Ammar, B. Krupczak, and K. Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. Technical Report GIT-CC-93/22, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, March 1993.

[27] P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. on Parallel and Distr. Syst.*, 1(1):17–25, Jan 1990.

[28] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Vienna, 1992.

[29] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(3):87–103, Dec 1993.

[30] S. Mishra, L. L. Peterson, and R. D. Schlichting. Experience with modularity in Consul. *Software Practice & Experience*, 23(10):1059–1075, Oct 1993.

[31] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, Dept. of Comp. Sci., Univ. of Arizona, June 1994.

[32] M. Olsen, E. Oskiewicz, and J. Warne. A model for interface groups. In *Proceedings of the 10th IEEE Symp. on Reliable Distributed Systems*, pages 98–107, Pisa, Italy, Sep 1991.

[33] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Trans. Comput. Syst.*, 10(2):110–143, May 1992.

[34] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Trans. on Softw. Eng.*, SE-14(1):30–37, Jan 1988.

[35] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, Aug 1989.

[36] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, Oct. 1984.

[37] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, June 1993.

[38] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Dept. of Computer Science, Aug. 1994.

[39] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *Proceedings of SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.

[40] R. Yavantkar. MCP: A protocol for coordination and temporal synchronization in multimedia collaborative applications. In *Proc. of the 12th IEEE Conf. on Distr. Computing Syst.*, page 606, Yokohama, Japan, June 1992.