COYOTE

A System for Constructing Fine-Grain Configurable Communication Services

Nina T. Bhatti   Matti A. Hiltunen   Richard D. Schlichting   Wanda Chiu

TR 97-12

Coyote: A System for Constructing Fine-Grain Configurable Communication Services[1]

Nina T. Bhatti   Matti A. Hiltunen   Richard D. Schlichting   Wanda Chiu

TR 97-12

Abstract

Communication-oriented abstractions such as atomic multicast, group RPC, and protocols for location-independent mobile computing can simplify the development of complex applications built on distributed systems. This paper describes Coyote, a system that supports the construction of highly modular and configurable versions of such abstractions. Coyote extends the notion of protocol objects and hierarchical composition found in existing systems with support for finer-grain objects called micro-protocols that implement individual semantic properties of the target service. A customized service is constructed by selecting micro-protocols based on their semantic guarantees and configuring them together with a standard runtime system to form a composite protocol implementing the service. Micro-protocols within a composite protocol can share data and are executed using an event-driven paradigm that enhances configurability. The overall approach is described and illustrated with examples of services that have been constructed using Coyote, including atomic multicast, group RPC, membership, and mobile computing protocols. A prototype implementation based on extending *x*-kernel version 3.2 running on Mach MK82 with support for micro-protocols is also presented, together with performance results from a suite of micro-protocols from which over 60 variants of group RPC can be constructed.

July 7, 1997

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the Office of Naval Research under grants N00014-91-J-1015, N00014-94-1-0015, and N00014-96-0207.

1 Introduction

Complex applications built on a distributed architecture can be simplified if the underlying software provides suitably tailored communication-oriented abstractions. 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 [BSS91, CASD85, MSMA90, PBS89]. Other abstractions of this type include group RPC [Che86, CGR88, Coo90], membership [Cri91, KGR91, MPS92], distributed transactions [BHG87], and protocols related to multimedia applications [KE93, Yav92]. Such abstractions logically form a *distribution support layer*—that is, a layer that provides execution guarantees regarding the distributed state of the application—that occupies a place in the system hierarchy between the application and traditional operating system functionality. From a networking perspective, they can be viewed as application-oriented network protocols or distributed services that are implemented at a high level in the protocol stack.

This paper describes Coyote, a system that supports the construction of highly modular and configurable versions of complex high-level protocols. Coyote extends the notion of *protocol objects* supported by systems such as ADAPTIVE [SBS93], Horus [RBM96], and the *x*-kernel [HP91] to finer-grain modules called *micro-protocols* that implement individual properties of the target service as separate modules. For example, with atomic multicast, one micro-protocol might implement the consistent ordering requirements, while another might implement reliable transmission. Micro-protocols can also be used to implement different semantic variants of the same property. For example, there may be multiple micro-protocols implementing different message-ordering or retransmission policies. A service is then configured from a suite of micro-protocols representing a range of possible semantic choices based on the particular execution guarantees needed for the given application. Micro-protocols are structured using events and event handlers, which enhance configurability by minimizing explicit references between modules.

Our approach provides a number of advantages over a monolithic approach to constructing similar services or systems that support only coarse-grain hierarchical composition. These include:

- *Configurability*. With fine-grain semantic-based configurability, customized services can be constructed that provide execution guarantees that are targeted to the specific requirements of the application.

- *Efficiency*. With a customized service, the application can avoid execution overhead that often results from inclusion of unnecessary properties in monolithic services, as well as choose the most efficient alternative given the current execution environment.

- *Reusability*. The same collection of micro-protocols can be used to construct multiple related services for different applications rather than having to implement entirely new services from scratch.

- *Extensibility*. New execution properties can be added as needed by writing additional micro-protocols and including them in an existing micro-protocol suite.

- *Expressibility*. The approach provides a new, more general model for structuring protocol objects that allows micro-protocols implementing a given service to communicate using events or shared data as appropriate.

The need for a new approach has been reinforced by experience with existing systems. For example, using the *x*-kernel to construct Consul, a protocol suite implementing atomic multicast [MPS93a, PBS89], highlighted deficiencies in several areas, including support for complex interactions among protocol objects, limited facilities for data sharing, and an orientation towards hierarchical protocol composition at the expense of more flexible combinations [MPS93b]. The developers of xAMP, a real-time atomic multicast protocol, report a similar experience [Fon94, VRB89]. The lack of a sufficiently rich protocol object interface has also been cited as one of the motivations for developing Horus, a successor to the Isis system [RBM96]. In addition, our approach is related to recent work in configurable operating systems [BSS+95, EKO95, HPM93, MMO+94] and on the use of object-orientation and reflection to build customizable operating systems [MHM+95, Yok92] and services for distributed systems [FNP+95].

Two prototype versions of Coyote have been constructed, including one that augments the *x*-kernel's standard hierarchical object composition model with the ability to internally structure *x*-kernel protocol objects using micro-protocols. 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* that implements the desired service. 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. Internally, the framework implements an event-driven execution paradigm, in which event handlers within micro-protocols are executed whenever events for which they are registered—for example, message arrival or a timeout—occur [HS93]. 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). The second prototype is written in C++. It demonstrates the generality of the approach and serves as another platform for prototyping composite protocols.

This paper makes several contributions. First, it describes a new approach to designing complex high-level protocols based on fine-grained modules and the event-driven execution paradigm supported by Coyote. Second, it gives examples of a number of services that have been developed using this paradigm, including group RPC, membership, and protocols for mobile computing. Finally, it describes the design and performance of the *x*-kernel based prototype, which executes on a collection of DecStation 5000 workstations running Mach MK82 and *x*-kernel version 3.2. A suite of 18 micro-protocols from which over 60 semantic variants of group RPC can be configured is used to illustrate the prototype system.

2 Software Architecture

The fundamental components of our approach are composite protocols, micro-protocols, events, and a runtime framework. This section describes these aspects of Coyote, presents an example micro-protocol that implements local detection of site failures from a membership suite, and discusses various design issues.

2.1 Composite Protocols and Micro-Protocols

A composite protocol is a software module that realizes the functionality of a service such as atomic multicast or group RPC within the distribution support layer of the system. Composite protocols are composed hierarchically with other composite protocols or regular network protocols such as UDP to form the *network subsystem* with which the application interacts. The interface of the composite protocol is determined by the system being used to perform the hierarchical composition, which is prototype-specific. For example, in the *x*-kernel prototype, the interface is the collection of operations supported by the *x*-kernel's UPI, including message `push()`, `pop()`, `demux()`, and control operations. In the C++ prototype, the interface is not fixed *a priori*, but rather is specified by the programmer of the service.

Composite protocols are constructed from micro-protocols that are selected based on the desired properties of the final service. Micro-protocols within a given composite protocol can share variables,

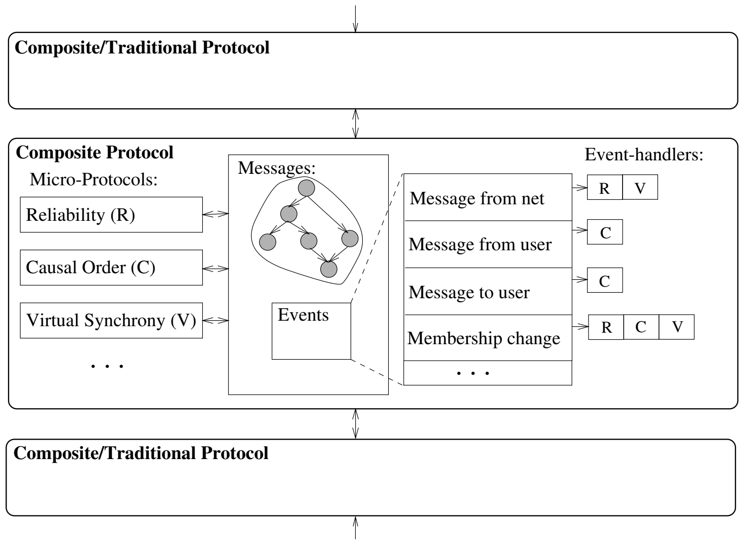

Figure 1: Composite protocol.

and include event handlers that are executed when events for which they are registered occur. The runtime system (framework) manages execution and implements the event mechanism. The composition of micro-protocols into a composite protocol is done statically—that is, the code is linked together with the framework at the time the network subsystem is built—although the execution patterns and specific properties being enforced can be changed at runtime by rebinding events and handlers. Modifying the collection of micro-protocols that comprise a composite protocol at runtime is discussed in section 6.2.

A protocol stack containing a composite protocol that illustrates this approach is depicted in figure 1. In the middle is the composite protocol, 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 event handlers within micro-protocols that are to be invoked when the event occurs.

A micro-protocol is logically structured as shown in figure 2. In addition to event handlers, it supports local variables that can be accessed only within the micro-protocol, exported procedures that can be invoked from within other micro-protocols, local procedures, and initialization code that is executed at system startup time. We express micro-protocols in this paper using an informal pseudo-code, although in reality, they are written in C or C++ depending on the prototype used. It would be straightforward to define a Protocol Definition Language (PDL) and implement the requisite translator to enforce scoping and modularity rules if desired.

2.2 Events and Handler Execution

Execution activity within a composite protocol is initiated when an event occurs, which causes event handlers registered for that event to be executed. An event is either a *predefined event*, in which case it is detected and raised implicitly by the framework, or a *user-defined event*, in which case it is detected by a micro-protocol and raised explicitly using a triggering mechanism. Predefined events correspond to state changes in composite protocols that are common across most communication services, and are typically related to messages. These include arrival of a message at the composite protocol from the application (i.e.,

```
micro-protocol mpname {
  /* Declaration of local variables */
      ...

  /* event handlers */
    handler hname(...) {
      code for hname
    }
    ...

  /* local procedures */
    procedure pname(...) {
      code for pname
    }
    ...

  /* initialization */
    initial {
      initialization code
    }
  }
```

Figure 2: Micro-protocol schema

higher-level protocols), arrival of a message from the network (i.e., lower-level protocols), and departure of a message in either direction. The predefined events used in the *x*-kernel prototype are given in section 3.2.3.

When an event is raised, a thread is allocated from a pool maintained by the framework to execute each handler bound to that event.[1] The execution model is logically multi-threaded, so that multiple handlers—either associated with the same or different event occurrences—may in general be executed concurrently. The execution of handlers associated with an event occurrence can, however, be serialized by specifying that the event is *sequential* when the event is defined initially using a call to the framework. Such semantics can simplify micro-protocol code in certain cases when execution order is important, such as when a subsequent event handler depends on a side effect caused by a previously-executed handler. However, it has the disadvantage of leading to implicit dependencies between micro-protocols, which can adversely affect configurability.

Execution of a micro-protocol that raises a user-defined 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. Execution of predefined events raised by the framework is asynchronous in the sense that the framework continues executing after allocating threads to execute handlers associated with detected events.

Five types of operations are defined to manipulate events within micro-protocols:

- *Creation:* Define a new event and its attributes (i.e., concurrent or sequential).
- *Bind:* Associate a handler with an event so that the handler is invoked when the specified event occurs; remains in effect until unbound.

[1] In most cases, this process is optimized in the implementation to use a single thread to avoid allocation overhead; see section 3.

- *Unbind:* Remove the association between a handler and an event.
- *Raise:* Trigger an event, passing specified values as argument values to the registered handlers.
- *Cancel*: Cancel further handler invocations associated with the same event occurrence.

Unbinding and cancellation are especially useful for dynamically altering handler execution [Hil96]. Note also that there are no restrictions on binding handlers to events, so that a given handler may be registered for multiple events and multiple handlers may be registered for a single event.

Finally, the model supports *timer events* that are generated after a specified interval of time has passed rather than any particular action of a micro-protocol or framework. Timer events can either be *single* or *periodic*; the former are generated only once, while the latter are generated repeatedly. Timer events can also be deallocated, which prevents the event from being generated if the interval has not yet passed and cancels future occurrences in the case of periodic events. Deallocation is atomic with respect to handler execution, so that all handlers are allowed to complete before deallocation if any handler has already begun execution.

2.3 Framework

The framework is a runtime system that implements the event mechanism and provides facilities for shared data that can be accessed by all micro-protocols within the composite protocol. It also implements the composite protocol interface, which enables it to interoperate with other protocols in the protocol stack.

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 [MPS93a, PBS89]. 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 protocol stack. *Micro-protocol attributes* contain micro-protocol-specific information about the message. For example, a reliability micro-protocol might keep private state information about the message indicating whether it was acknowledged or is being retransmitted and by which hosts. In addition, attributes are used to build headers for messages that leave 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 enters 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.

2.4 Coordinated Message Sending and Deallocation

Facilities are provided that allow multiple micro-protocols to coordinate when to pass a message either up or down the protocol stack as appropriate. To see the need for this, consider a message that has arrived from the network via a lower-level protocol. An acknowledgment micro-protocol may dispatch a reply message to acknowledge receipt, which completes processing of the message as far as it is concerned. However, an ordering protocol that places strict requirements on message ordering may wish to force the message to stay in the composite protocol. Realizing such coordination is especially difficult since it must function correctly for any combination of micro-protocols.

To support this type of coordination, the framework uses *send bits* associated with a message. There is one bit per micro-protocol configured into the composite protocol, and when all send bits have been set, the framework automatically passes the message to the next protocol in the appropriate direction. A bit can be defined to be on by default in cases where a micro-protocol does not need to restrict when a message exits the composite protocol.

Similarly, *deallocate bits* allow micro-protocols to coordinate message deallocation. Again, each micro-protocol has a unique deallocate bit for each message. When all deallocate bits are set, the framework raises a “Message ready to be deallocated” predefined event, which gives micro-protocols the opportunity to free any information related to the message. To avoid dangling memory references, actual deallocation is deferred until after all outstanding send operations referencing the specified message have completed. This coordinated message deallocation is a configuration-time option, which allows use of a centralized scheme if more convenient. For example, in some composite protocols, a single micro-protocol can determine when a message can be deallocated based on its role in spooling messages to stable storage

2.5 An Example Micro-protocol

To illustrate the programming style and event-driven paradigm, figure 3 presents a failure detection micro-protocol from a configurable process group membership service that provides consistent information about which sites are functioning and which have failed at any given time [HS97]. In this service, micro-protocols are provided that implement different variants of multiple properties, including site failure detection, agreement among sites on changes, partition handling, and consistent ordering of membership change notification relative to other membership changes and application messages. The suite is based on a token-passing paradigm, in which the sites are organized into a logical ring and a token is circulated to pass information, and to implement ordering and agreement.

The micro-protocol shown in figure 3 implements a live variant of site failure detection.[2] The micro-protocol uses a global data structure `Membership`, which is the agreed group membership, and maintains a global data structure `SuspectList`, which is a list of suspected membership changes represented as pairs, {`site_id, FAILURE or RECOVERY`}. It notifies other micro-protocols about suspected failures by updating the `SuspectList` and then raising either the event `Suspect_Next_Down`, which indicates the failure of the site's successor in the logical ring, or event `Suspect_Change`, which indicates the failure of any other site. These events would then typically be fielded by other micro-protocols.

Most of the code in the micro-protocol is used to implement three complementary techniques for detecting a potential site failure. The first, as mentioned above, is to suspect a failure if a site does

[2] The names of the event handling operations are taken from the *x*-kernel prototype; see section 3.3.

```
micro-protocol LiveFailureDetection(Limit:int, check_period:real) {
  var SilentList:  set of int;     /* list of sites not heard of lately */

  handler handle_no_ack(var site:int, attempts:int) {
    if (attempts < Limit and (site, FAILURE) ∉ SuspectList)
      attempts++;
    else {
      SuspectList += (site, FAILURE); attempts = 1;
      raiseEvent(Suspect_Next_Down, SYNC, 1, site);
    }
  }

  handler monitor() {
    for each m:int ∈ Membership do {
      if (m ∈ SilentList and (m, FAILURE) ∉ SuspectList) {
        SuspectList += (m, FAILURE);
        raiseEvent(Suspect_Change, ASYNC, 2, m, FAILURE);
      }
      SilentList += m;
    }
  }

  handler handle_msg(var msg:NetMessage) {
    if (msg.type != JOIN) then {
      if ((msg.sender, FAILURE) ∈ SuspectList)
        SuspectList -= (msg.sender, FAILURE);
      if (msg.sender ∈ SilentList) SilentList -= msg.sender;
    }
    elseif (msg.sender∈Membership and (msg.sender,FAILURE) ∉SuspectList) {
      SuspectList += (msg.sender, FAILURE);
      raiseEvent(Suspect_Change, ASYNC, 2, msg.sender, FAILURE);
    }
  }

  handler handle_membership_change(msg:ApplMessage) {
    if (msg.type == FAILURE) SuspectList -= (msg.changed, FAILURE);
  }

  initial {
    for each m:int ∈ Membership do SilentList += m;
    addEventHandler(Forwarding_Failed, handle_no_ack);
    addEventHandler(Msg_Inserted_Into_Bag, handle_msg);
    addEventHandler(Membership_Change, handle_membership_change);
    createEvent(Liveness_Check, 0);
    setTimerEvent(Liveness_Check, PERIODIC, check_period, 0);
    addEventHandler(Liveness_Check, monitor);
  }
}
```

Figure 3: **LiveFailureDetection** micro-protocol

not acknowledge receipt of the token. This method is implemented in handler `handle_no_ack`, which is activated by the `Forwarding_Failed` event raised by the token-handling micro-protocol when no acknowledgment is received within a specified time. If the attempt to forward the token fails `Limit` times, the site in question is added to `SuspectList` and `Suspect_Next_Down` is raised.

The second technique is to suspect a failure if no messages are received from a site within a specified period of time. This method is implemented by handlers `handle_msg` and `monitor`, which use the local data structure `SilentList` to maintain the sites from which no message has been received during the time period `check_period`. In this scheme, `SilentList` is initialized to all sites and `handle_msg` removes a site when a message is received from that site. In particular, `handle_msg` is executed whenever the `Msg_Inserted_Into_Bag` event is raised, which occurs when a message arrives from a lower-level protocol and is inserted into the composite protocol's shared bag of messages. The `monitor` handler raises the `Suspect_Change` event for every site that remains in `SilentList` at the end of `check_period` time units and then reinitializes `SilentList` for the next period. `monitor` is executed at the appropriate times by binding the handler to a periodic timer event `Liveness_Check` that is created and activated using the `createEvent` and `setTimerEvent` operations in the micro-protocol initialization code.

The third technique is to suspect a failure if a message of type JOIN sent by a site currently in the membership arrives from a lower-level protocol. The receipt of such a message indicates that the sender of the message is recovering and wants to join the group, which implies that it necessarily must have failed given the crash failure model used. This method is also implemented as part of the `handle_msg` handler.

The final handler in the micro-protocol, `handle_membership_change`, updates `SuspectList` when an agreed membership change is made. This is indicated by the event `Membership_Change`, which is triggered in another micro-protocol once agreement is reached.

Further details of the group membership service are given below in section 5.2. In addition, a complete description of a configurable group RPC service and its performance using the *x*-kernel based prototype is described in section 4.

2.6 Design Issues

Designing a configurable service requires addressing a number of issues, ranging from identifying relevant properties to developing a set of micro-protocols that maximizes configurability. The overall process is illustrated in figure 4. The first two boxes represent abstract characterizations of the service, while the final

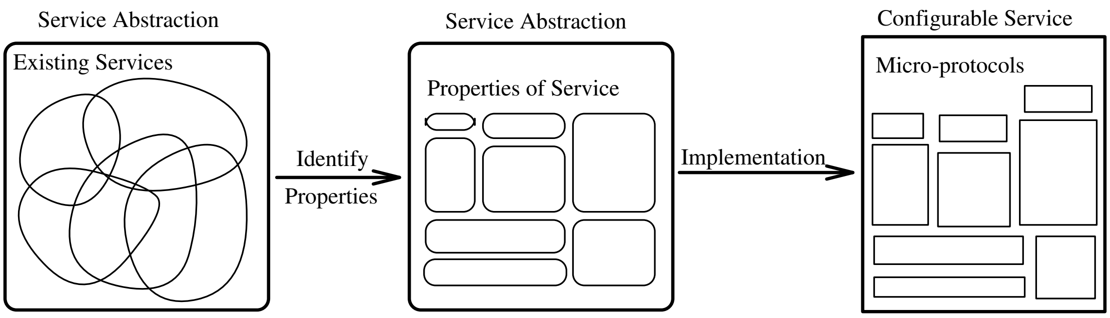

Figure 4: Constructing a configurable service

box represents the micro-protocol suite making up the final configurable service.

In this paper, we focus on the implementation aspect of the process, and specifically, on developing micro-protocols given abstract properties. Techniques for identifying and formally specifying properties are beyond the scope of this paper, but are addressed in [Hil96, HS95b] for the case of membership services.

In building micro-protocols for a service, a basic initial issue is selecting an overall implementation strategy for a service. For example, the suite may be based around a centralized strategy using a coordinating site or may be fully decentralized. Issues to be considered include ease of implementation, efficiency in terms of number of messages or execution time, and applicability of the strategy to the underlying computing environment. For example, a broadcast based strategy may not be the best choice if the underlying network does not provide hardware broadcast facilities. Existing implementations of the service often provide good alternatives for the general implementation strategy.

A second issue is designing the micro-protocols to maximize configurability, since, in general, micro-protocols cannot be combined in arbitrary ways to yield a valid service. We can identify four relations between micro-protocols that affect configurability:

- *Conflict*. Micro-protocols m_1 and m_2 conflict if they cannot be configured into the same system, either because the corresponding properties are impossible to guarantee at the same time or because of design decisions made during the implementation.
- *Independence*. Micro-protocols m_1 and m_2 are independent if m_1 can be used without m_2, m_2 can be used without m_1, and m_1 and m_2 can be used together, where the combination guarantees both the properties implemented by m_1 and m_2.
- *Dependence*. Micro-protocol m_1 *depends on* micro-protocol m_2 if m_2 must be present in the configuration of a service and operate correctly in order for m_1 to provide its specified service. In practice, this means that if m_1 is to be configured into a service, m_2 must be configured in as well.
- *Inclusion*. Micro-protocol m_1 *includes* micro-protocol m_2 if m_1 implements a property strictly stronger than that implemented by m_2 without relying on m_2 being configured in the service. In practice, this means that m_1 and m_2 would be redundant if configured together into a service.

The relations may result from fundamental relations between the properties being implemented or from implementation choices made during the design process. For example, failure detection cannot be both live and accurate in asynchronous systems [FLP85, SM94], so micro-protocols implementing these variants conflict based on their underlying properties. On the other hand, the relation between a micro-protocol implementing consistent total message ordering and one implementing causal ordering depends on the implementation design, since it is possible to implement total order either with or without an underlying causal order (e.g., [MPS93a] and [MMSA+96], respectively).

Similarly, whether two micro-protocols have a dependency or inclusion relation is often based on implementation convenience. For example, causal message ordering also trivially realizes FIFO ordering and could be built on a FIFO ordering micro-protocol. However, the knowledge that messages are already FIFO ordered does not simplify the implementation of causal order, so it is easier to implement causal order independent of FIFO. This leads to an inclusion rather than dependence relation between the respective micro-protocols. In other situations, the implementation of a property can take advantage of the guarantees provided by some other micro-protocol. For example, the implementation of message ordering properties can exploit an atomicity micro-protocol that ensures that all sites will eventually receive every message. This leads to the dependency relation between micro-protocols.

The relations between micro-protocols and the underlying abstract properties can be represented graphically using configuration and dependency graphs, respectively [Hil96]. These graphs can be used to evaluate if a proposed configuration is feasible, as well as to enumerate all possible valid configurations.

3 *x*-kernel Prototype

3.1 Introduction

The *x*-kernel prototype of Coyote is based on version 3.2 of the *x*-kernel, which runs as a user-level task on Mach version MK82. Written in C, the prototype is structured as a collection of library routines that are linked with the user-written micro-protocols to create a composite protocol. The composite protocol is then included in the *x*-kernel protocol graph in the normal way. The *x*-kernel was selected as the implementation environment because of its efficient message handling, novel thread execution architecture, ease of configuration and modification, and portability. The only non-*x*-kernel facilities used beyond normal C language library routines are three Mach functions for thread scheduling and management.

The hardware platform used for the experiments reported here consists of DecStation 5000/240s connected by a 10 Mb Ethernet. These systems are based on MIPS R3000 micro-processor running at 40 MHz with a separate off-chip 64 KB instruction and data caches, and 16 MB of memory. Coyote and its associated micro-protocol suites are currently being ported to a Pentium-based cluster running OSF/RI MK 7.2 and CORDS, the *x*-kernel augmented with *paths* for resource allocation [TMR96].

The prototype differs somewhat from the model outlined in the previous sections, most importantly by not supporting event cancellation and allowing event creation only during the initialization phase of a composite protocol. A second C++ prototype that illustrates the generality of the approach is briefly discussed below in section 6.1.

3.2 Framework

3.2.1 Uniform interfaces.

The framework encapsulates micro-protocols and delivers messages to and from other *x*-kernel protocols. Externally, the framework provides the standard *x*-kernel interface operations such as `call()`, `push()`, `pop()`, and `demux()`. This allows 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 either a synchronous call interface or an asynchronous push interface to accommodate both styles of *x*-kernel protocols. For a push interface, the reply message (if any) is returned asynchronously.

3.2.2 Thread management

The multiple threads of control used for executing event handlers are implemented using *x*-kernel threads, which are implemented in turn using C-threads in Mach. Using *x*-kernel threads rather than spawning C-threads directly has a number of benefits. For example, it makes the threads visible to the *x*-kernel, which permits the programmer to use the built-in *x*-kernel features for doing execution monitoring and debugging, simplifying the programming process. It also allows the system to exploit the *x*-kernel's optimized thread management. In particular, the *x*-kernel preallocates a pool of C-threads at initialization time and manages them directly, which avoids the overhead of thread creation when an event is raised.

When an event is raised, a thread is allocated from this pool to execute each associated handler. Given the specialized nature of thread support in the *x*-kernel, this is actually done by scheduling an *x*-kernel timing event to execute with a zero second delay, which upon expiration places the allocated thread on the ready list to be scheduled with other C-threads. No new threads are allocated for events that are raised synchronously; rather, the same thread that triggered the event will sequentially execute all handlers registered for the event prior to returning. Sequential events are treated the same way.

The protocol writer can choose to have event handler invocations be implemented by procedure calls rather than threads even in the case when the event is raised asynchronously. This optimization is targeted for sequential machines where a procedure call is typically more efficient than spawning a thread, and is used as the default given our hardware configuration. 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.

Threads that enter a composite protocol are treated differently than the normal *x*-kernel paradigm, where a single thread typically shepherds a message through the entire protocol stack. In this model, a thread arriving at a protocol can either continue up or down the protocol graph by executing a `push()` or `pop()`, can terminate (e.g., if the message is discarded), or can be blocked within the protocol (e.g., to wait for a reply). However, in a composite protocol, entering threads are typically disassociated from their messages and may be used to execute event handlers. The specific strategy used depends on whether handler execution is implemented by threads or procedure calls. With the former, the incoming thread is usually terminated once other threads are activated to execute pending event handlers, while with the latter, the thread is blocked to realize synchronous raise semantics. Threads may also be blocked instead of terminated if a matching reply is expected. Threads that exit the composite protocol are reassociated with messages to restore the normal *x*-kernel model.

3.2.3 Messages

The main data structure in a composite protocol is a bag of messages that is accessible to all micro-protocols. Messages in the bag, referred to as *CP Messages*, are network messages augmented with additional attributes that micro-protocols use to share per-message data. CP messages are based on *x*-kernel messages, which are optimized for common message manipulation functions such as header pushes and pops, fragmentation, and assembly. In addition to the normal *x*-kernel message operations, additional operations are provided for efficiently accessing attributes. As noted above, the framework also provides for coordinated control of message attributes, creation of headers and attributes, sending, and deallocation.

Bag of Messages. The following are the key operations provided for manipulating the bag:

- `CPMsg = newItem(xMsg, direction)`: Allocates and initializes a new CP message, returning a pointer to the appropriate structure; `direction` indicates whether the message is traveling up or down through the *x*-kernel protocol graph.
- `insertItem(CPMsg)`: Inserts `CPMsg` into the bag; triggers the `Message_Inserted_Into_Bag` event.
- `deleteItem(CPMsg)`: Removes `CPMsg` from the bag, although the actual storage deallocation is done under micro-protocol control; triggers the `Message_Deleted_From_Bag` event.
- `setSendBit(ProtocolID, CPMsg)`: Sets the send bit for micro-protocol `ProtocolID`; when all bits are set, the `Message_Ready_To_Be_Sent` event is triggered.
- `setDeallocateBit(ProtocolID, CPMsg)`: Sets the deallocation bit for micro-protocol `ProtocolID`; when all bits are set the `Message_Deallocate` event is triggered.

Other predefined events supported by the prototype are `Message_Popped_To_CP`, which indicates that a message has been popped to the composite protocol (CP) by a lower-level protocol; `Message_Popped_From_CP`, which indicates that a message has been popped from the CP to a higher-level protocol; `Message_Pushed_To_CP`, which indicates that a message has been pushed to the CP by a higher-level protocol; and `Message_Pushed_From_CP`, which indicates that a message has been pushed from the CP to a lower-level protocol.

Attributes and Headers. CP message attribute values are often derived from information contained in message headers, such as the sender id, destination id, and message type. To aid this function, all micro-protocol suites must include a single `header_to_attribute` procedure that sets attributes based on header values and localizes header format knowledge to one procedure. Typically, this procedure is called by a verification micro-protocol after the incoming message has been validated. Similarly, when a message is about to be sent, the message header is normally constructed from the attributes. The `attribute_to_header` procedure is called by the framework during a send message operation for this purpose.

Coordinated Message Sending. When a message is created with `newItem`, all the send bits are initially off. A micro-protocol sets its corresponding bit with `setSendBit(CPMsg, ProtocolID)` using the protocol id assigned to the micro-protocol at system initialization time. When all send bits are set, the CP Message is ready to be sent and the `Message_Ready_To_Be_Sent` event is raised by the framework. The send bits restrict sending of messages both in the upward direction (to applications) and downward direction (to the network). If a micro-protocol is not directly involved in the decision on when to send a message, its bit is set on by default.

Sending Out of Band. Although coordinated sending is the expected norm, there are occasions when a particular micro-protocol might wish to send a message without another micro-protocol's interference or knowledge. This is accomplished with `sendMessageOutOfBand(CPMsg)`, which sends the message without raising any events.

Coordinated Deallocation of Messages. As described above, garbage collection in a composite protocol can be implemented either by a single micro-protocol or as a function coordinated among multiple micro-protocols using deallocation bits. With the latter approach, the `Message_Ready_To_Be_Deallocate_Message(CPMsg)` event is raised when all deallocate bits have been set. The handlers for this event then perform the actual mechanics of message deallocation and deletion, i.e., calling `deleteItem()` and then freeing memory. The user chooses the style of deallocation support that is desired by setting a C preprocessor variable that activates the deallocation-based events and bits.

3.3 Events and Handler Execution

All events are implemented using the same two types of data structures, an *event description structure* and a collection of *event invocation structures*. For a given event, the first contains pointers to the handler functions registered for this event, handler names, and the number of event parameters, and is passed as the event descriptor in every `raiseEvent` call. Event invocation structures, on the other hand, are allocated when an event is raised and contain argument values and other invocation-specific data. One such invocation structure is created for each handler registered for the triggered event. In addition to these two structures, timer events require additional structures to record the state of the event and the current execution status to support cancellation and repetition.

The following operations are provided for manipulating events and event handlers:

- `event = createEvent(eventName, numArgs)`. Allocates and initializes a new event definition structure and returns a handle that is used for later operations. `eventName` is a descriptive string naming the event, and `numArgs` is the number of argument values passed to handlers when the event is raised. Used for both regular and timer events.

- `addEventHandler(event, handler, handlerName)`. Binds `handler` to `event` by adding a function pointer to the list of handlers for `event`. The ordering of the add operations determines the execution order for sequential events. `handlerName` is a descriptive name for the handler.

- `deleteEventHandler(event, handler)`. Removes `handler` from the list of handlers for `event`.

- `raiseEvent(event, type, numArgs, arg1, arg2, arg3, arg4, arg5)`. Triggers `event` and passes the specified `numArgs` argument values to handlers. `type` is `SYNC` if the event is to be executed synchronously or `ASYNC` if executed asynchronously. Used by micro-protocols for triggering user-defined events and by the framework for triggering predefined events.

- `event_invoke = setTimerEvent(event, type, interval, numArgs, arg1, arg2, arg3, arg4, arg5)`. Sets a timer event to execute after `interval` microseconds have elapsed. `type` specifies whether the timer event is periodic or once-only. `numArgs` and the argument values are as above. `event_invoke` is a handle used in the cancel and detach functions.

- `outcome = deallocTimerEvent(event_invoke)`. Deallocates a timer event. The return value indicates whether the deallocation was successful, or whether the handlers associated with the event had already started to execute.

- `detachTimerEvent(event_invoke)`. Instructs the framework to deallocate structures when handler execution for the event completes; if a timer event will never be canceled, then the structure can be detached immediately after the timer is set.

One issue that had to be addressed in designing the event mechanism was the restriction that the *x*-kernel timing events used to implement thread management (section 3.2.2) support only a single function accepting a single argument. To circumvent this restriction, the timing event is passed a "super handler" procedure that, when invoked as a result of the timer expiring, unpacks arguments from the event invocation structure and passes them to a specified handler. This process is repeated for each handler bound to the event. Thus, the super handler acts as a procedural wrapper around handler executions, recording the start and termination of each handler, and handling memory allocation and deallocation.

To maintain uniformity, synchronous event execution uses the same event invocation structure to pass arguments even though *x*-kernel timing events are not used in this case. Instead, the super handler is invoked directly as a procedure, which provides a synchronous call style with blocking semantics. The super handler is the same as in the asynchronous case, and, in fact, is unaware whether it was called from an *x*-kernel event or directly as a procedure. Asynchronous event execution is also performed in the same way if the procedure call optimization is being used.

3.3.1 Bounding call depth

Stack overflow and unfair scheduling can occur if event handlers are executed using procedure calls either because an event is raised synchronously or because asynchronous execution is being optimized as procedure calls. The problem arises due to nested events, which occur when a handler executing as a result of one event occurrence triggers another event. With procedure call execution, the nested event is executed prior to other events, which can cause stack overflow in the executing thread if the nesting goes very deep. In addition, events raised outside the calling chain—indicating arrival of a new message, for

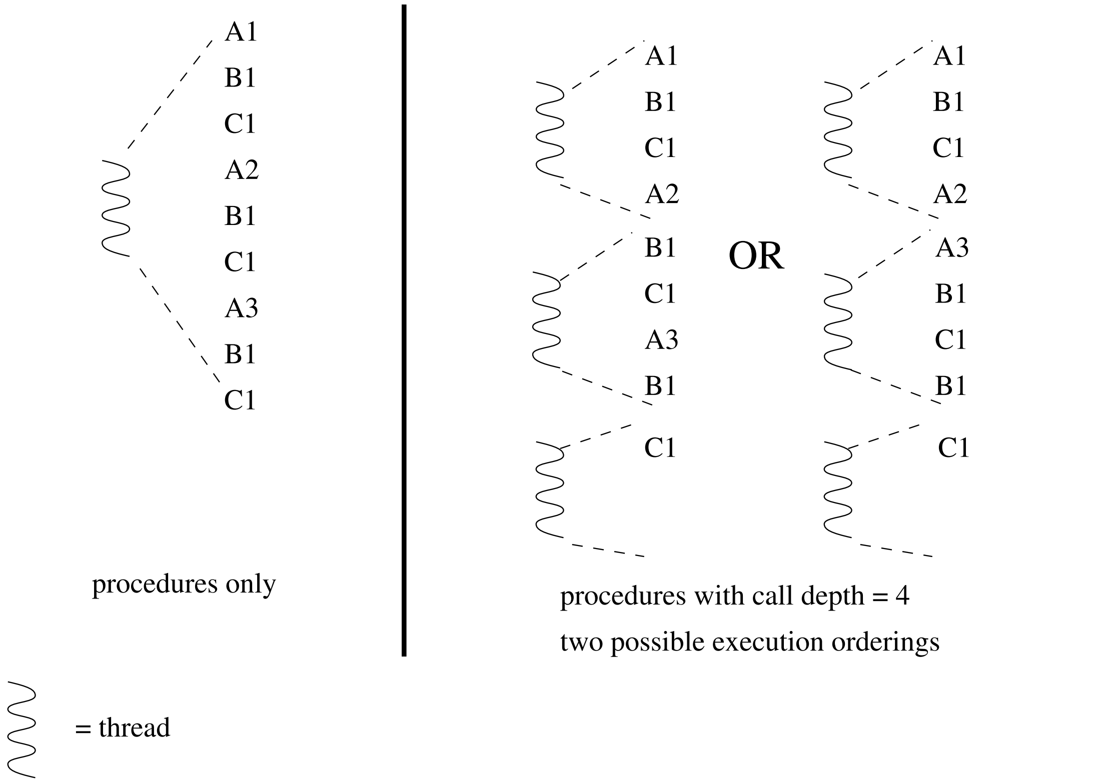

Figure 5: Possible event handler executions with and without call depth bounding.

example—may go unserviced for long periods of time, leading to starvation and other undesired behavior in certain types of composite protocols.

To eliminate these problems, *call depth bounding* can be enabled. This option creates a hybrid execution scheme in which handlers are executed as procedure calls only until a specific nesting depth has been reached. At that point, the next asynchronous trigger is executed by a new thread and the current thread is allowed to terminate, thereby completing the call chain. The current call depth is recorded by associating a *call depth count* with each C-thread. This count in incremented each time the thread invokes an event handler and decremented each time it returns.

As an example, consider an event A with three handlers A1, A2, A3. Each of these handlers raises event B, which has one handler B1. B1 in turn raises event C with one handler C1. All events are raised as "ASYNC" events and the framework is configured to execute events with procedure calls. The execution order in this case is A1 B1 C1 A2 B1 C1 A3 B1 C1, as shown in the left panel of figure 5. However, the scenario changes if the framework is configured with a call depth of four. A1 B1 C1 A2 would be executed using procedure calls by a single thread as before, but this thread would then terminate, and B1 and A3 and would be scheduled asynchronously to execute in an arbitrary order. If B1 executes first, then the rest of the execution order would be B1 C1 A3 B1 C1. The situation is analogous if A3 executes first. The right panel of figure 5 shows the execution structure with call depth bounding enabled.

3.3.2 Reducing handler execution overhead

As already noted, the prototype allows the use of procedure calls in lieu of thread allocation to reduce the overhead of executing handlers. Although not yet implemented, two other possible optimizations that would reduce this overhead further are in-lining of handlers and support for *handler guards*. In-lining involves replacing calls to the framework `raiseEvent()` procedure with the micro-protocol code using appropriate compiler support. The compiler would enforce visibility rules and rename variables in the handler code that clash with variables in the surrounding micro-protocol code.

A handler guard is a boolean expression associated with a handler at registration time that controls under what conditions the handler is executed. In particular, the expression is evaluated by `raiseEvent()` prior to executing a handler when the triggering event occurs. If the result is false, the handler is not executed, thereby saving the cost of spawning a thread or executing a procedure call. The rationale for this feature is that many handlers are structured to first check a condition and then only continue if the condition holds. For example, almost all micro-protocols register a handler for the `Message_Inserted_Into_Bag` event, but most are concerned with only a specific message type. Such a condition could be specified as a guard, potentially reducing the number of handler invocations substantially.

3.4 Measurements of Event Implementation Performance

Event invocation and handler execution are the heart of the composite protocol, and therefore, the efficiency of events is central to the performance of the system. As discussed above, there are two implementations of events that can be used: lightweight user-level threads or procedure calls. We considered both styles and compared the performance and runtime behavior of each implementation using the experimental platform described in section 3.1.

The relative cost of using procedure calls versus a thread-based implementation was assessed using a null composite protocol designed to measure event execution times. Each test measured the round trip message transmission time based on 1000 round trips for two processes. The first test configuration 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 (CP-T). Figure 6 illustrates the structure and message flow of the second and third configurations.

The results are shown in Table 1. 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 observed to be low.

3.5 Creating a Composite Protocol

Source files are used to structure the components of a composite protocol. There are three categories of files: user-supplied, user-modifiable, and read-only, as follows:

- *User-supplied files*. Contain micro-protocol code, required routines such as attribute-to-header, header-to-attribute, attribute printing, and initialization code.

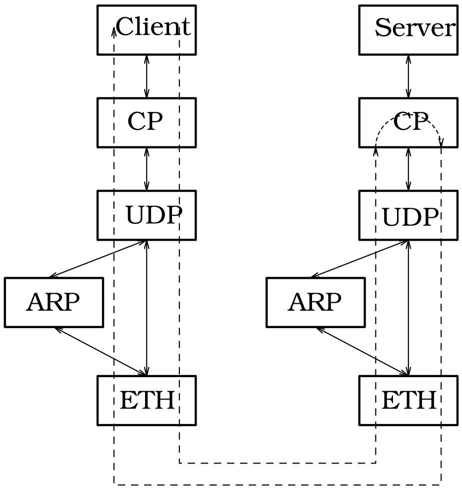

Figure 6: Experimental configuration

- *User-modifiable files*. Supplied by the system, but can be modified to customize the service further. For example, the *x*-kernel push and pop procedures can be customized for multicast or other sending styles. The user can also modify defines to configure the composite protocol for call style interface or push style, enable procedure call execution of asynchronous event execution, bound call depth, and enable deallocation bit support. Some of the possible modifications require *x*-kernel specific knowledge, but the default setup should be sufficient for most micro-protocol suites.

- *Read-only files*. Contain only framework-specific code and are not alterable. These files include standard functions, such as bag of messages routines, event support, and the *x*-kernel encapsulation protocol.

One user-modifiable file concerns the lower-level protocol used. By default, a composite protocol uses UDP, which is sufficient for any composite protocol that only requires unreliable datagram service. However, the user has the option of changing the lower-level protocol to any *x*-kernel protocol, perhaps even another composite protocol. To do this, a support file must be created that contains procedures to create participant addresses and manage communication channels built on the new lower-level protocol sessions. The selection of the lower-level protocol also naturally affects the selection of micro-protocols

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 1: Roundtrip time for null CP (in msec)

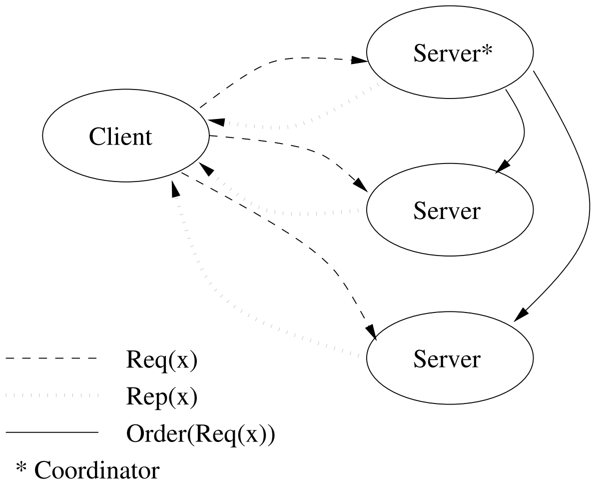

Figure 7: Process and message architecture.

in the composite protocol. For example, if the lower-level protocol is an unreliable multicast protocol, the send routine in a composite protocol implementing atomic multicast can be much simpler since the lower-level protocol can issue a message to each group member automatically.

4 A Group RPC Service

This section presents the design and implementation of a collection of micro-protocols implementing different variants of regular and group RPC (GRPC) using the Coyote *x*-kernel prototype. Micro-protocols are configured together into a composite protocol called `Group_RPC`, which is then incorporated into an *x*-kernel protocol graph with UDP as its lower-level supporting protocol. Measuring the performance of `Group_RPC` therefore yields the relative cost of the different configurations and their underlying semantic properties.

Our version of GRPC is based on point-to-point messages, so clients send individual requests to each server group member. Figure 7 shows the process level architecture and message flow between clients and servers. Request messages, Req(x), are sent from clients, while servers send reply messages, Rep(x), back to the client. The reply messages are used to create a return value, which is passed back to the application once the RPC terminates. If total order is included, then one server acts as a coordinator that determines the ordering of requests. Thus, for each request, the coordinator sends an ordering message, Order(Req(x)), to all other servers.

4.1 Micro-protocols

The micro-protocol suite is based on the semantic variations of GRPC described in [HS95a]; the categories that follow represent semantic variations of termination, ordering, communication, collation, call style, membership, and failure handling.

4.1.1 Termination semantics

Termination semantics specify the guarantees that are given about the termination of a call. Included in the client composite protocol.

- `BOUNDED (BND)`. Provides for bounded termination of client requests, i.e., either the request is executed within some interval or an exception is returned. When a request is sent, a timer event is set to generate a timeout.
- `UNBOUNDED(UBND)`. No *a priori* bound is set on a client request, so the client may wait indefinitely for a response.

4.1.2 Ordering semantics

Ordering semantics determine what guarantees are given about the execution order of requests by servers. If none of the micro-protocols are included, any ordering is possible.

- `FIFO`. Forces FIFO ordering of requests from each client at a server. Servers order client requests using sequence numbers that are assigned by clients. Requests from multi-threaded clients are serialized before transmission. Included in both clients and servers.
- `TOTAL`. Forces total ordering of all client requests at all servers. The ordering of client requests is determined by a designated server process that acts as a coordinator. All non-coordinator servers receive requests but do not execute them until an ordering message is received from the coordinator. The coordinator delays sending a reply to the client until the request is deliverable at at least one other server—i.e., all earlier messages in the ordering have been received—which ensures a correct ordering even if the coordinator fails. Included only in servers.
- `FIFO` and `TOTAL`. Total order that preserves FIFO ordering. Realized by including both FIFO and TOTAL micro-protocols.

4.1.3 Communication semantics

Communication semantics specify guarantees about the communication between the client and server. Reliable transmissions are guaranteed if acknowledgment and retransmission micro-protocols are both included.

- `ACK`. Acknowledges request and response messages, and handles timeouts. If an acknowledgment message is not received in time, an event is raised notifying other micro-protocols for possible retransmission. Included in clients and servers.
- `RETRANSMIT (RET)`. Sends retransmission requests for missing messages and responds to retransmission requests. Included in clients and servers.
- `CONTROL_RETRANSMISSION(CRET)`. Only used with totally ordered communication. Sends acknowledgments and waits for acknowledgment of control messages between servers. Included in servers only.

4.1.4 Collation semantics

Collation semantics specify how responses from the server group are combined and the result returned to the client. All micro-protocols included only in clients.

- ONE_ACCEPT (1ACC). Implements a policy of accepting the first reply from any server as satisfying the client's request. Other responses are ignored.
- ALL_ACCEPT (AAC). Implements a policy of collecting replies from all functioning servers before the RPC call is completed. If a server is no longer functioning, the new membership is used to prevent waiting forever for a response from a failed server.

Of course, it would be easy to define micro-protocols that wait for responses for any number in between as well.

4.1.5 Call semantics

Call semantics specify whether the call thread in the client blocks. All micro-protocols are included only in clients.

- SYNC. Provides synchronous request/reply call-style interface. The call thread is blocked until the call completes.
- ASYNC. Provides asynchronous push-style interface. The result of the call is returned by an upcall.

4.1.6 Membership semantics

Membership semantics specify how information is collected about failed and functioning processes, and what can be guaranteed about the correctness of this information. Since point-to-point messages are used, clients must maintain information about server group membership to send requests and collate responses.

When total ordering of messages is used, server groups must also maintain their own membership to determine if all messages are received and executed by all servers. A single server failure is tolerated at any given time, including the coordinator; recoveries and addition of new servers are not currently supported. When membership does change, the change event is consistently ordered at each server so servers agree when the change occurred.

- CLIENT SERVER MEMBERSHIP (CSMEM). Manages the server membership for a client. Using the ACK micro-protocol, a client times out unresponsive servers and removes them from the server list. This membership list is used by the ALL_ACCEPT micro-protocol to determine when all responses are received and for sending point-to-point messages to all servers. CSMEM is required for all configurations and is included in clients.
- SERVER MEMBERSHIP (SMEM). Manages the server membership list for members of a server group. Membership is initialized at boot time from a static list, which is then updated dynamically as servers fail. The server with the largest host address is the coordinator, so no negotiation is required to determine the coordinator. Included in servers.
- LIVE. Servers send liveness messages to each other in a ring topology to detect when a server fails. If no liveness message is received within the interval, then the member is suspected to have failed and the "suspect host dead event" is raised. This triggers the SIM agreement micro-protocol (below) to confirm the failure. Included in servers only.

- SIMPLE AGREEMENT(SIM). Simple agreement sends "server is dead" messages to other servers if a "suspect host dead" event is triggered. All other servers simply accept this declaration of a defunct server even if they have information to the contrary. Included in servers only.
- VIRTUAL_SYNCHRONY(VS). Virtual synchrony ensures that membership change messages appear in the same order relative to data messages for all hosts. When a failure occurs, non-failing servers exchange information about the highest deliverable request that has been received before the failure occurred. This allows servers to synchronize on what messages should have been received before the membership change occurred. Included in servers only.

4.1.7 Failure semantics

Failure semantics specify what guarantees are given to the client about the execution of requests by the server.

- UNIQUE. Eliminates duplicate request messages using sequence numbers. Ensures that a request is never executed more than once by forwarding the original reply message if a duplicate request is received. Included in servers only.

4.1.8 Driver micro-protocol

The suite requires a driver micro-protocol GRPC for all combinations of micro-protocols. Verifies incoming messages and maintains client and server state information. Required for clients and servers.

4.2 Combining Micro-Protocols

There are 64 possible GRPC configurations given the above collection of micro-protocols. The composite protocol may have synchronous or asynchronous call style, bounded or unbounded calls, one accept or all accept collation, and 8 selections of orderings. Figure 8 illustrates the possible selections of micro-protocols. All configurations require the GRPC and CSMEM micro-protocols.

The selection of call style, bounding of calls, and collation policies are independent choices and each only requires the inclusion of one micro-protocol implementing that property. Unique execution and FIFO ordering are each achieved through the inclusion of one micro-protocol. Reliable transmission of messages is accomplished through acknowledgment and retransmissions of messages, which requires the ACK and RET micro-protocols.

Total ordering is complex and requires several micro-protocols, because the servers must maintain their own membership to ensure totally ordered execution of client requests. As already noted, all servers receive request messages and one server acts as the coordinator, generating ordering messages that guide all servers to complete the requests in total order. All servers must receive all the request messages, so servers maintain their membership through the use of a liveness micro-protocol, LIVE. When a server is suspected of having failed, a simple agreement micro-protocol, SIM, is executed, which causes all servers to delete failed servers from membership lists. Virtual synchrony, VS, is used to ensure that the membership change occurs at the same point with respect to the stream of request and ordering messages. Servers must communicate reliably or communication would halt if an ordering message was lost. This functionality is provided by CRET.

Formal dependency and configuration graphs for a similar suite of group RPC micro-protocols can be found in [HS95a, Hil96].

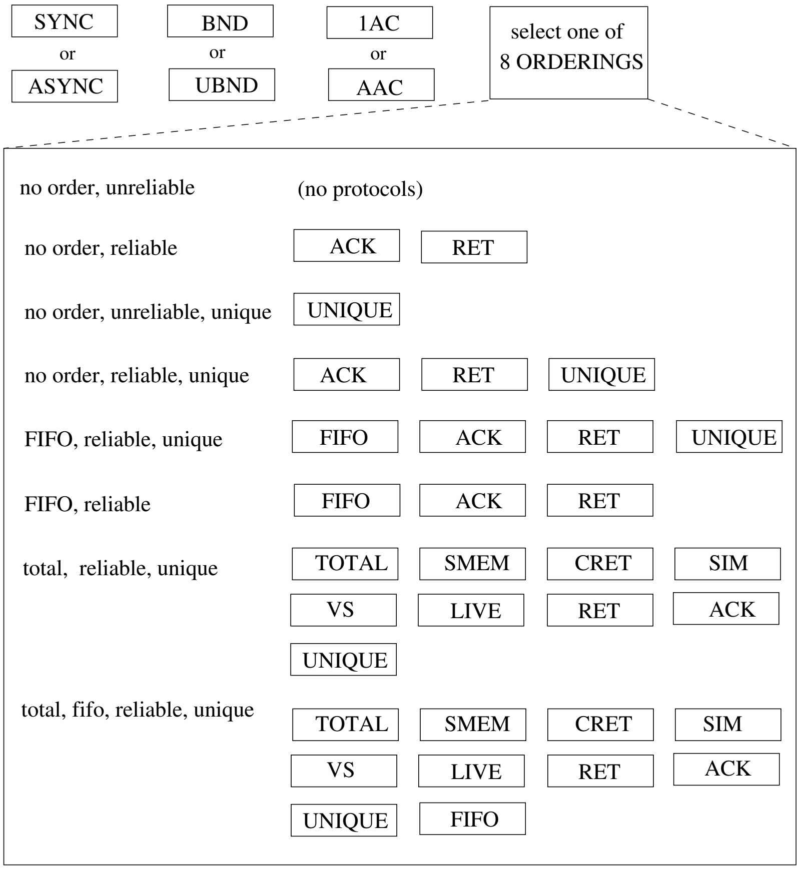

Figure 8: Group RPC configuration selections.

4.3 Performance Measurements

Tests consisted of one or more clients 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 round trip 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`.

All measurements were done on the experimental platform described in section 3.1. Tests requiring three or less hosts execute server and client processes on DecStation 5000/240s. For tests requiring more than three hosts, all server processes execute on DecStation 5000/240s and client processes execute on DecStation 5000/200s. Like the DecStation 2000/240s, DecStation 5000/200s are MIPS R3000 microprocessor based systems with separate off-chip 64 KB instruction and data caches, and 16 MB of memory. However, the DecStation 5000/200's processor clock rate is 25 MHz instead of 40 MHz.

The average roundtrip times for the various configurations are given in Table 2. The relative ordering is what one would expect: normal Sun RPC using the *x*-kernel (BL) is fastest, and for the same microprotocol configurations, increasing the number of servers and clients results in increased execution time. 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 collation policies, group

	System Configuration	Clients	Servers	avg	var
BL	*x*-kernel Sun RPC	one	one	4.38	0.00035
C1	GRPC,SYNC,1AC,CSMEM,UBND	one	one	6.30	0.018
		one	two	8.82	0.032
C2	GRPC,SYNC,AAC,CSMEM,UBND	one	two	8.85	0.052
C3	GRPC,ASYNC,FIFO,1AC,CSMEM,UBND	one	one	5.68	0.024
C4	GRPC,ASYNC,FIFO,1AC,CSMEM,BND	one	one	6.12	0.019
C5	GRPC,ASYNC,1AC,CSMEM,BND	one	one	5.58	0.012
C6	GRPC,ASYNC,AAC,CSMEM,BND, RET,ACK	one	one	8.49	0.026
		two	two	16.59	0.849
		two	three	22.71	0.008
C7	GRPC,ASYNC,FIFO,AAC,CSMEM,BND, UNIQUE,RET,ACK	one	one	8.91	0.043
		two	two	19.68	0.018
		two	three	23.76	0.003
C8	GRPC,ASYNC,AAC,CSMEM,BND, UNIQUE,RET,ACK,SMEM,LIVE,SIM CRET,TOTAL,VS	one	one	10.62	0.077
		two	two	35.22	0.230
		two	three	48.47	0.224
C9	GRPC,ASYNC,FIFO,AAC,CSMEM,BND, UNIQUE,RET,ACK,SMEM,LIVE,SIM CRET,TOTAL,VS	one	one	10.64	0.219
		two	two	44.19	2.40
		two	three	50.66	0.83

Table 2: Time for `Group_RPC` call (in msec)

membership, multiple servers, or message ordering options needed for more complex applications.

In general, increasing the guarantees the communication service provides results in a slower roundtrip execution time. This is as expected, since the more guarantees that are given, the more expensive the algorithms required to implement the communication service. However, micro-protocols that increase message traffic degrade performance more than micro-protocols that only add computation time to clients or servers. For example, adding FIFO to configuration C8 (measured in configuration C9) results in a small increase in timing (0.02 msec) because it only adds sequence numbers to requests. On the other hand, the difference between configuration C7 and C8 is the addition of server membership and total ordering. The timing difference between these tests is appreciable (1.71 msec), because configuration C8 increases message traffic between servers. Similarly, the increase in the number of servers for configurations using total order results in large increases in running time since message traffic grows quadratically with the number of servers. For configurations C8 and C9, we can also see large increases (10 msec) with the addition of another server.

Finally, note that each of the nine tested configurations for `Group_RPC` has reasonable classes of applications for which the realized semantics are appropriate. For example,

- Configuration C3 is useful for serializing requests from multithreaded clients since it provides FIFO ordering.

- Configuration C6 could be used for a reliable name service providing information about host utilization and resource availability since it provides reliable transmission between clients and servers.
- Configuration C7 adds unique execution to reliable communication, which is useful for transmitting non-idempotent operations where multiple execution could give incorrect results.
- Configuration C9 provides rich semantics including FIFO-preserving total ordering, which makes it an ideal base for building financial services that are executed on a cluster of servers for fault-tolerance.

Using our approach, then, a developer can fine-tune the semantics of the underlying communication platform by constructing a custom version of group RPC, thereby incurring a cost only for those properties actually required.

5 Other Services

In addition to group RPC, a number of other configurable services have been designed using Coyote, including an atomic multicast service, a group membership service, and communication support for mobile computing. This section provides an overview of these three micro-protocol suites.

5.1 Atomic Multicast

The atomic multicast service is a customized version designed for the runtime system of a fault-tolerant version of the Linda coordination language [ACG86] called FT-Linda [BS95]. Linda is a language for parallel programming based on tuple space (TS), a communication abstraction defined as a bag that can hold data elements called tuples. Processes use TS to communicate and synchronize by depositing and withdrawing tuples from a TS. However, Linda as originally defined does not address fault tolerance, so the crash of a participating host may lead to loss of a portion of the TS. To address this issue and provide for atomic execution, FT-Linda extends the original model with *stable tuple spaces* and support for atomic execution of sequences of TS operations. Stable tuple spaces in particular are implemented by replicating the TS on all participating sites and using the replicated state machine approach to maintain consistency [Sch90]. This requires a communication substrate that provides totally ordered atomic multicast, failure detection, and membership services for the process group implementing the TS.

The Coyote micro-protocol suite realizing these requirements consists of a dozen micro-protocols implementing the following:

- *Membership*. Raises suspicion of failure based on timeouts, and implements an agreement protocol in which agreement on a suspected failure is reached only if all remaining sites suspect the failure.
- *Reliability*. Implements reliable point-to-point message transmissions using negative acknowledgments.
- *Ordering*. Ensures that messages are delivered in a consistent total order on all sites.
- *Stability*. Maintains information about whether a message is stable, i.e., known to be have been received by all functioning sites.

The last three use an approach similar to that used in Psync [PBS89] and the Consul system [MPS93a]. In particular, messages sent to the progress group are stored in a *context graph* that maintains causality information about messages i.e., the predecessors of a message m are those messages that the sender of m received before sending m. The ordering micro-protocol maintains this graph and implements the causal ordering, while the stability micro-protocol accesses the graph to determine stability information. The reliability micro-protocol uses the predecessor information in messages to detect that a message has not been received and to request a retransmission.

The set of micro-protocols interact through shared data structures and events. The most important shared data structures are the context graph and a membership list of participants. The set of events includes those representing the different phases of message transit through the composite protocol, as well as events for dealing with predecessors, stability, and membership changes. For example, the ordering micro-protocol raises event `PredecessorsNeeded` when a message arrives before its causal predecessors, and the reliability micro-protocol raises `PredecessorsReceived` when the required messages arrive.

5.2 Group Membership

A group membership service is logically a software layer that generates messages indicating changes in the membership of a process group and forwards them to higher levels, including the application. These *membership messages* can report, for example, failures, recoveries, or the joining of two partitions. Given such a system, the properties of a membership service can be defined in terms of what membership messages it generates and when they are delivered to the application [HS95c].

For the purposes of constructing a micro-protocol suite, we divided the properties of a membership service as follows:

- *Change detection*. Includes variants involving the tradeoff between *accuracy*—a change is reported only if the change has indeed occurred—and *liveness*—all changes are eventually reported.
- *Agreement*. Describes how membership sets on different sites relate to one another at any point in time. Variants include *agreement on membership messages*—all sites in the same partition deliver the same set of membership messages—and *eventual agreement on membership views*—the membership sets of sites in the same partition eventually converge.
- *Ordering*. Specifies how delivery of membership messages is ordered with respect to one another and/or application messages. Variants include *total order*—membership messages (only) are delivered in the same order on sites in the same partition—and *virtual synchrony*—membership message are delivered to the application at the same point relative to the application message stream at all sites in the same partition.
- *Partition handling*. Specifies the behavior of the service when (logical) partitions occur, during partitioned operation, and when partitions join.

The micro-protocol suite consists of 25 micro-protocols that implement multiple variants of each property, including seven ordering properties and four for partition handling. The design is based on a token-passing paradigm in which a token is circulated around a (logical) ring to collect and distribute information. The token has one entry for each membership change being processed, with each entry having a number of different fields used by different micro-protocols. Around 20 different events are used for interaction. Most of the 25 micro-protocols implement abstract properties, although a few provide basic functionality on which others build. For example, a `TokenDriver` micro-protocol handles the mechanics of passing the token between the sites and dealing with lost tokens. The micro-protocols can be configured to realize over 1000 semantically different composite protocols [Hil96].

The complete suite of micro-protocols has been implemented and tested using the C++ prototype (see section 6.1). A subset has also been ported to the *x*-kernel prototype.

5.3 Protocols for Mobile Computing

To illustrate the value of the approach for other types of applications, a micro-protocol suite has been developed to support customization of communication protocols for mobile computing [Bha96]. This suite supports customization of the underlying communication software for different types of mobile hosts, for the base stations that function as gateways between the wired network and mobile host, and for stationary agent processes that act as proxies on the wired network for mobile hosts. The goal is to allow customization in two areas:

- *Handoff*. Different approaches are provided for transferring communication responsibility between base stations as a mobile host moves.
- *Quality of Service (QoS)*. A conceptual framework is provided for negotiating QoS guarantees for connections to mobile hosts, such as throughput, jitter, latency, and packet retransmissions limits. Also supports renegotiation of guarantees as network conditions change or as the mobile host changes location.

Each variant is implemented by separate micro-protocols for the mobile host, base station, and agent, as appropriate.

Micro-protocols related to handoff are divided into orthogonal behaviors governing when a handoff is needed or desirable, the actual handoff procedure, and disconnection from the old base station. For handoff detection, the variants are:

- *ICMP messages*. Base stations periodically transmit an ICMP (Internet Control Message Protocol) message requesting that mobile hosts within range identify themselves with a response. A host that arrived since the last transmission is a candidate for a handoff.
- *Host beacons*. Similar to above, except that mobile hosts periodically transmit messages automatically rather than in response to ICMP messages.
- *Monitor-based*. Relies on lower-level hardware or software to detect conditions that might necessitate a handoff.
- *Lazy detection*. Optimization in which a handoff is only performed if application-level communication activity has occurred recently even if the mobile host has moved.

For effecting the handoff of a mobile host from one base station to another, four options are provided that differ in where the final authority for the handoff resides:

- *Request/reply*. The new base station makes an explicit handoff request to the old base station, who replies yes or no.
- *Negative acknowledgment*. The new base station broadcasts a handoff notification to all base stations; the new station assumes responsibility unless some other station replies no.
- *Mobile host initiated*. The mobile host controls the handoff by transmitting information to the new base station that allows it to contact the old station and effect the handoff.

- *Agent coordinated*. A mobile host's proxy controls the handoff by deciding on requests from base stations to perform a handoff.

In addition, a separate micro-protocol is provided that prevents oscillation when a mobile host is within range of two or more base stations.

Finally, disconnection determines how an old base station disposes of packets destined for delivery to a mobile host that has been handed off. Three approaches are provided:

- *Drop*. Packets are simply dropped, with the expectation that a higher-level protocol will later retransmit them via the new base station.
- *Forward*. Packets are forwarded from the old base station to the new one for delivery.
- *Drain*. An attempt is made to transmit quickly as many packets as possible to the mobile host, with the remaining packets being dropped. Based on the observation that old base stations can often still communicate with a mobile host for a period of time even after a handoff.

Combining the micro-protocols implementing different variants for detection, handoff, and disconnection taking dependencies into account gives 45 possible variations dealing with handoffs.

QoS is implemented by micro-protocols that can be configured into a composite protocol in addition to those dealing with handoff. Rather than implement all combinations of actual policies, our approach provides a software structure—micro-protocol "templates," in essence—that encapsulate the actual policies. These templates implement the event-based interaction with other micro-protocols in the same composite protocol, as well as the system-wide interaction needed to negotiate and renegotiate QoS attributes.

This collection of micro-protocols can be used to built any number of custom communication services for mobile computing, including those with semantics similar to existing systems such as Crosspoint [CR94, CLR95], DataMan [AB93, BBIM93], InfoPad [LSBR94, LBSR95], and PARC TAB [AGSW93, STW93]. For example, Crosspoint combines ICMP for handoff detection, negative acknowledgments for effecting the handoff, and forwarding for disconnection, while PARC TAB combines a beacon strategy, agent-based handoff, and packet dropping. InfoPad includes QoS guarantees as well. The ability to configure multiple systems from a single collection of micro-protocols simplifies development, as well as promotes experimentation with new combinations and customized solutions.

6 Discussion

6.1 C++ Prototype

To confirm the generality of the model and provide another prototyping environment, a version of Coyote has also been implemented in C++. The basic system consists of approximately 1000 lines of code and uses the standard thread package in Sun Solaris to implement event handling and other aspects of the runtime system. The multiple sites of a distributed architecture are simulated within a single address space, which allows precise control over execution parameters such as the number of sites, the message transmission times, and failure rates. The system model provided by this prototyping environment is an asynchronous system where sites experience crash failures and messages may be lost.

The major component of the system is a service layer that implements composite protocols and micro-protocols as C++ classes `CompositeProtocol` and `MicroProtocol`, respectively. `CompositeProtocol` contains the runtime system, implementing the event-driven execution model and providing operations to register event handlers and trigger events. It also implements a common interface for composite protocols with operations that allow protocols above and below to transfer messages. Each specific service such as RPC or membership is realized as a derived class of `CompositeProtocol` by defining the service-specific events, shared data structures, and initialization. `MicroProtocol` is the base class from which the micro-protocols implementing a specific service are derived. The prototype has been used to implement the group membership service described in section 5.2, as well as a second version of group RPC.

In addition to serving as a prototyping platform, this version illustrates the ease with which the event-driven approach can be implemented using standard operating system facilities. The system could also be used as the starting point for a true distributed implementation based on C++ using, for example, CORBA [OMG95a, OMG95b] for underlying distribution support. Most of the existing runtime system and micro-protocol code would carry over with minimal changes in this case.

6.2 Dynamic Reconfiguration

While Coyote is primarily designed for configuring a service from micro-protocols at system build time, the underlying execution model is well-suited for dynamically altering execution-time behavior. Among other things, such a capability is useful for constructing *adaptive systems* that can modify their behavior in response to changes in the environment. Such changes could be, for example, processor or link failures, changes in communication patterns or frequency, changes in failure rates or types, or changed user requirements.

One technique for altering execution-time behavior in Coyote is to rebind events to different event handlers to change the code that is executed when events occur. This, in essence, allows micro-protocols already configured into the composite protocol to be activated to replace existing micro-protocols, leading to a change in execution behavior. For example, a new message ordering algorithm could be activated in an atomic multicast service should user requirements change, or a change from a negative to a positive message acknowledgment scheme could be made if application traffic falls below some threshold. Of course, in many cases, such changes require coordination among sites [HS96].

A second more powerful technique is to change the actual micro-protocols within a composite protocol at runtime. This can be done, for example, using active networking techniques to introduce the needed changes, and runtime compilation to make the actual code modifications. Of course, event rebinding of the type outlined above would also be required to activate the new code. Such an approach is more complex, but has the potential to increase the flexibility of the system dramatically. For example, it would allow new services to be introduced to a site—either locally or from across the network—and installed without disrupting operation, thereby supporting long-term evolution of the software. It would also allow these middleware services to be reconfigured to deal with extreme changes in the execution environment that were not anticipated when the original system was built.

The current Coyote *x*-kernel prototype supports dynamic rebinding of events and handlers, but not on-line introduction of new micro-protocols. Implementing and using the latter is the subject of on-going research.

6.3 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 [OOW91] and the work on multicast reported in [Gol92]. In contrast to these, our approach is more general and provides more flexibility for the designer. Object-oriented structuring and the use of reflection to alter behavior in fault-tolerant distributed programs or general operating systems are described in [AS94, FNP+95, Yok92]. As demonstrated with the C++ prototype, our approach can also be mapped into an object-oriented paradigm, but is general enough to be implemented using more traditional approaches. Also in the area of fault-tolerance, a number of papers describe abstract properties of services or certain components such as failure detectors [Bla91, CT96, SR93], or present families of related services [CASD85, WS95]. Such work is complementary to that presented here since it suggests how configurable versions of a service can be implemented using micro-protocols.

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. Many of our goals related to system customization, code reuse, and protocol configurability are adopted from the *x*-kernel. Horus [vRHB94, RBM96] and Ensemble [BRV96] have been used to construct configurable versions of distributed services, although the models are limited to stack-like configurations of coarse-grained protocol objects. Armada [ASJS96] and OSF/RI MK [TMR96] extend the *x*-kernel model to support real-time, but with the same protocol object and composition model.

Other *x*-kernel related work has explored the use of finer-grain protocol objects [OP92], 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 [Rit84] also supports modularization of protocols, but its model is also hierarchical and relatively coarse-grained.

Somewhat closer to our work is the ADAPTIVE system [SBS93], 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 system divides operation of any given service into predefined phases (e.g., for transmission control), each with alternative algorithms implemented as separate modules. Interactions between the modules in this model are predefined and fixed, in contrast with the more flexible possibilities of our model.

Several other efforts have concentrated on supporting parallel execution of modular protocols, including [GNI92, LAKS93]. 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 services of the type described here.

Finally, recent work on new generation operating systems has emphasized similar customization goals, but in a more general context. These include the Exokernel [EKO95], Scout [MMO+94], and SPIN [BSS+95], as well as work on subcontracts [HPM93] and application-controlled file caching [CFL94]. 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 the type of services considered here. Moreover, the configurability they provide is typically more coarse-grained than our approach, which emphasizes choice among specific semantic properties of services constructed above the operating system.

7 Conclusions

Distributed services implemented as high-level protocols 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 the Coyote system for implementing configurable versions of these protocols in which fine-grained micro-protocol objects are composed using a runtime framework to yield a 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.

The *x*-kernel prototype demonstrates the feasibility of the Coyote model as a realistic approach for implementing high-level protocols. The prototype allows composite protocols to be composed transparently with standard *x*-kernel protocols in a protocol graph, with the arrival of messages from protocols above or below it in the graph generating events that result in handlers being invoked to deal with the message. The prototype can be configured to use procedure calls rather than thread allocation to optimize performance, and provides for call depth bounding to deal with scheduling issues that arise as a result. A C++ prototype confirms the generality of the approach and provides another environment for experimenting with configurable services.

Coyote has been used to develop several communication-related services, including group RPC, membership, atomic multicast, and protocols for mobile computing. Experiments conducted using the *x*-kernel group RPC suite clearly indicate the performance penalties associated with more complex semantics, which can be avoided using our approach by customizing the semantics to those needed by the application. The system offers other benefits as well, such as the ability to use a single configurable service across a wide variety of applications and a flexible programming model that encourages decoupling of orthogonal concerns. This approach also has the potential to support configurability of other types of execution attributes as well, such as those related to real time and security. Current research is exploring these issues with a goal of developing a single integrated approach to providing fine-grain configurability of such general Quality of Service (QoS) attributes for services constructed on distributed systems.

Acknowledgments

X. Han implemented a portion of the group RPC micro-protocol suite and ported a subset of the membership suite to the *x*-kernel prototype. D. Guedes designed and implemented the atomic multicast service for FT-Linda. X. Han provided comments that greatly improved the paper.

References

[AB93] A. Acharya and B. R. Badrinath. Delivering multicast messages in networks with mobile hosts. In *Proceedings of the 13th IEEE International Conference on Distributed Computing Systems*, pages 292–299, May 1993.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[AGSW93] N. Adams, R. Gold, B. Schilit, and R. Want. An infrared network for mobile computers. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 41–51, Aug 1993.

[AS94] G. Agha and D. Sturman. A methodology for adapting to patterns of faults. In G. Koob and C. Lau, editors, *Foundations of Dependable Computing: Models and Frameworks for Dependable Systems*, pages 23–60. Kluwer Academic Publishers, 1994.

[ASJS96] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. RTCAST: Lightweight multicast for real-time process groups. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 250–259, Jun 1996.

[BBIM93] B. Badrinath, A. Bakre, T. Imielinski, and R. Marantz. Handling mobile clients: A case for indirect interaction. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, October 1993.

[Bha96] N. Bhatti. *A System for Constructing Configurable High-Level Protocols*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Nov 1996.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[Bla91] A. Black. Understanding transations in an operating system context. *ACM Operating Systems Review*, 20(1):73–76, Jan 1991.

[BRV96] K. Birman, R. Renesse, and W. Vogels. The Ensemble distributed communication system. http://simon.cs.cornell.edu/Info/Projects/Ensemble/, 1996.

[BS95] D. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distr. Syst.*, 6(3):287–302, March 1995.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[BSS+95] B. Bershad, P. Savage, S.and Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, Colorado, Dec 1995.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th IEEE International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[CFL94] P. Cao, E. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, Nov 1994.

[CGR88] R. Cmelik, N. Gehani, and W. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing*, pages 55–61, Tokyo, June 1988.

[Che86] D. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM'86*, pages 406–415, Aug 1986.

[CLR95] D. Comer, J. Lin, and V. Russo. An architecture for a campus-scale wireless mobile internet. Technical Report CSD-TR95-058, Purdue University, Department of Computer Science, 1995.

[Coo90] E. Cooper. Programming language support for multicast communication in distributed systems. In *Proceedings of the 10th IEEE Conference on Distributed Computing Systems*, pages 450–457, Paris, France, 1990.

[CR94] D. Comer and V. Russo. Using ATM for a campus-wide wireless internetwork. In *Proceedings of the 1994 IEEE Workshop on Mobile Computing*, 1994.

[Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 34(1):225–267, 1996.

[EKO95] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, Dec 1995.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[FNP+95] J.-C. Fabre, V. Nicomette, T. Perennou, R. Stroud, and Z. Wu. Implementing fault tolerant applications using reflective object-oriented programming. In *Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing*, pages 489–498, Pasadena, CA, Jun 1995.

[Fon94] H. Fonseca. Support environments for the modularization, implementation and execution of communication protocols. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, June 1994. In Portuguese.

[GNI92] 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.

[Gol92] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Dept of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, 1992.

[Hil96] M. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, Jul 1996.

[HP91] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HPM93] 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.

[HS93] M. Hiltunen and R. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, Oct 1993.

[HS95a] M. Hiltunen and R. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th IEEE Conference on Distributed Computing Systems*, Vancouver, BC, May 1995.

[HS95b] M. Hiltunen and R. Schlichting. Properties of membership services. In *Proceedings of the Second IEEE Symp. on Autonomous Decentralized Systems*, pages 200–207, Phoenix, AZ, April 1995.

[HS95c] M. Hiltunen and R. Schlichting. Understanding membership. Technical Report 95-07, Department of Computer Science, University of Arizona, Tucson, AZ, Jul 1995.

[HS96] M. Hiltunen and R. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, 11(5):125–133, Sep 1996.

[HS97] M. Hiltunen and R. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 1997. To appear.

[KE93] 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, August 1993.

[KGR91] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, Wien, 1991.

[LAKS93] B. Lindgren, M. Ammar, B. Krupczak, and K. Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. In *Proceedings of the International Conference on Network Protocols*, March 1993.

[LBSR95] M. Le, F. Burghardt, S. Seshan, and J. Rabaey. InfoNet: the networking infrastructure of InfoPad. In *Proceedings of Compcon*, San Francisco, California, Mar 1995.

[LSBR94] M. Le, S. Seshan, F. Burghardt, and J. Rabaey. Software architecture of the InfoPad system. In *Proceedings of the Mobidata Workshop on Mobile and Wireless Information Systems*, Rutgers, New Jersey, Nov 1994.

[MHM$^+$95] K. Murata, R.N. Horspool, E. Manning, Y. Yokote, and M. Tokoro. Unification of active and passive objects in an object-oriented operating system. In *Proceedings of 1995 Int. Workshop of Object Orientation in Operating Systems (IWOOOS'95)*, Aug 1995.

[MMO$^+$94] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. In *Proceedings of the 1st Symposium on Operating Design and Implementation*, page 200, Nov 1994.

[MMSA+96] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, Apr 1996.

[MPS92] S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In J. Meyer and R. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Vienna, 1992.

[MPS93a] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(3):87–103, Dec 1993.

[MPS93b] S. Mishra, L. Peterson, and R. Schlichting. Experience with modularity in Consul. *Software–Practice and Experience*, 23(10):1059–1075, Oct 1993.

[MSMA90] P.M. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distr. Syst.*, 1(1):17–25, Jan 1990.

[OMG95a] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 1995.

[OMG95b] Object Management Group. *CORBAservices: Common Object Services Specification*, 1995.

[OOW91] 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.

[OP92] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[PBS89] L. Peterson, N. Buchholz, and R. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[RBM96] R. van Renesse, K. Birman, and S. Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[Rit84] D. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, October 1984.

[SBS93] 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.

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[SM94] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 138–147, Dana Point, CA, Oct 1994.

[SR93] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing*, pages 534–543, Toulouse, France, Jun 1993.

[STW93] B. Schilit, M. Theimer, and B. Welch. Customizing mobile applications. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 129–138, Aug 1993.

[TMR96] F. Travostino, E. Menze, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, Feb 1996.

[VRB89] 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.

[vRHB94] R. van Renesse, T. Hickey, and K. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Dept. of Computer Science, Aug 1994.

[WS95] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pages 106–115, Bad Neuenahr, Germany, Sept 1995.

[Yav92] R. Yavantkar. MCP: A protocol for coordination and temporal synchronization in multimedia collaborative applications. In *Proceedings of the 12th IEEE Conference on Distributed Computing Systems*, page 606, Yokohama, Japan, June 1992.

[Yok92] Y. Yokote. The Apertos reflective operating system: The concepts and its implementation. In *Proceedings of OOPSLA 1992*, pages 414–434, Vancouver, BC, Oct 1992.