A Reflective Middleware Framework for Communication in Dynamic Environments

Sebastian Gutierrez-Nolasco[1] and Nalini Venkatasubramanian[1]

University of California, Irvine, Irvine, CA 92697-3430 USA
{seguti,nalini}@ics.uci.edu

Abstract. The goal of a flexible communication framework is to allow dynamic customization of complex communication protocols without compromising the overall system performance. However, the communication protocols provided in successful middleware frameworks are usually tailored to specific and/or static requirements are not suitable for dynamic environments. In dynamic environments, applications must be able to customize communication protocols on-the-fly in order to respond to change requirements while protecting the system from reaching inconsistent states that can lead to deadlocks, livelocks and incorrect execution semantics. Hence, there is a need to identify interprotocol and protocol-(middleware) service interactions in order to ensure safe flexibility of the system. The work described here proposes a reflective communication framework (RCF) capable of expressing different levels of dynamic customization of communication protocols while ensuring safe customization of communication protocols.

1 Introduction

Current advances in storage and networking technologies such as wireless communication, mobile computing and real-time system support have enabled a new class of applications that require ubiquitous access to information, anywhere, anyplace and anytime. Such applications demand a high degree of flexibility and adaptability in order to deal with (a)changes in application requirements and (b)changes in the computational and communication environment. In order to ensure cost-effective system performance and provide a high degree of flexibility and adaptability in such environments, dynamic communication customization is required. In this paper, we develop a customizable communication framework where interactions between application components can be dynamically reconfigured to adapt to changing application, network and system conditions.

Middleware abstracts communication services, allowing the development of network transparent distributed applications. Although current middleware platforms provide mechanisms to enhance application behavior transparently at runtime [30], communication services provided are tailored to specific and usually static requirements. Such static communication frameworks are not well suited in dynamic asynchronous environments, where the communication framework must be able to automatically reconfigure itself in order to respond to changes

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2002, LNCS 2519, pp. 791–808, 2002.

in the computational and communication environments, such as resource and service restrictions, or changing network conditions and/or network availability.

For example, financial applications use time sensitive information to forecast and represent the stock market accurately. These applications usually run in private networks, which guarantee certain degree of reliability and security. In order to be able to access this information in a ubiquitous fashion (i.e. anywhere, anytime) via a handheld device, we require a communication environment that provides support for continuous flow of time sensitive information in the presence of varying network and system conditions. In order to achieve this, we assume that every message has a predefined valid time span, which defines the message lifetime. This time span is defined by the application and may vary over time. Thus, the communication framework will try to deliver the message before reaching the end of the time period by using a timed delivery protocol with priorities timeouts and selected retransmission based on the application time span. If the environment becomes also insecure, a secure communication protocol is required to guarantee message integrity, secrecy and possibly authenticity. Intuitively, layering the timed delivery protocol on top of the security protocol will not work, because the protocol was not designed to deal with an insecure medium, which may corrupt or fake messages. For instance, the header of the protocol will remain unprotected and may be modified. However, layering the security protocol on top of the timed delivery protocol may not work either since encryption is an intensive computational task that may significantly alter the message time span.

When two or more communication protocols are composed in order to obtain their combined benefits, their guarantees (desirable properties, e.g. security and timed delivery) are not always preserved. Preservation of protocol guarantees may crucially depend on the composition order and their interaction parameters.

Furthermore, the presence of middleware services may further complicate the communication between them. e.g. object mobility is used to exploit locallity and provide a certain degree of fault tolerance, but it also adds location uncertainity, which may impact the performance and correctness of the communication framework.

In this paper, we propose a reflective communication framework (RCF) using a meta-architectural approach to allow for dynamic customization and reconfiguration of communication protocols. The RCF is based on a formal model that allows for us to reason about the correctness of customizability. Specifically, we analyze issues with inter-protocol interactions, protocol-service interactions and show how multiple communication protocols and middleware services can be safely and dynamically reconfigured using the proposed architecture.

Our objective is to build an efficient framework that realizes the formal model and preserves the correctness properties. Furthermore, the basic RCF architecture can be integrated into generic distributed computing environment or middleware framework. To illustrate the feasibility of this approach, we have integrated the RCF model into a reflective middleware architecture, CompOSE|Q [24], currently being developed at the University of California, Irvine.

The remainder of this paper is organized as follows. In Section 2, we discuss the basic theoretical model of a flexible distributed environment. In Section 3, we describe the basic RCF framework. Composability and interaction issues are discussed in Section 4. We describe implementation issues encountered while developing an RCF and discuss initial performance results in Section 5. We describe related work and conclude in Section 6 with future research directions.

2 The Two Level Meta Architectural Model

Since the actor model of computation [11] incorporates the notion of encapsulation and interaction only via message passing, it offers a clear, flexible and simple semantic approach to describe distributed systems based on incoming communications [12]. The system is modeled as a group of self contained and independent autonomous objects, called actors, which communicate via asynchronous (buffered) message passing. On receiving a communication, an actor processes the message in a manner determined by its current behavior. As a result, the actor may: (1) Create new actors, (2) Change its behavior and (3) Send messages to itself or to other (existing) actors. Since mail addresses may be communicated in messages the configuration of the communication is dynamic and the activation order (one message activates another if the latter is sent during the processing of the former) determines communication patterns.

The two level actor machine (TLAM) model refines the actor model to specify, compose and reason about resource management services in open distributed systems[28]. In the TLAM model, a system is composed of two kinds of actors, base actors and meta actors, distributed over a network of processing nodes. Base actors carry out application level computation, while meta actors are part of the run-time system, which manages system resources and controls the run-time behavior of the base level. Meta actors communicate with each other via message passing as do base actors, and they may also examine and modify the state of the base actors located on the same node. Base level actors and messages have associated run-time annotations that can be set and read by meta actors. The annotations are invisible to base level computation. Actions which result in a change of base level state, are called events and meta actors may react to them if they occur on their node.

A TLAM configuration has a set of base and meta level actors and a set of undelivered messages. The actors are distributed over the TLAM nodes. Each actor has a unique name (actor identifier) and the configuration associates a current state to each actor name. The undelivered messages are distributed over the network (some are traveling along communication links and others are held in node buffers).

CompOSE|Q [24] is a a reflective middleware framework based on the TLAM model that facilitates the specification and reasoning about the dynamic composition and concurrent execution of resource management policies in open distributed systems. To ensure non-interference and manage the complexity of reasoning about components of open distributed environments in general, the

TLAM strategy is to identify key system services where non-trivial interactions between the application and system occur, i.e. base-meta interactions. We refer to these key services as *core services* (see Figure 1). Core services are then used in specifying and implementing more complex activities within the framework as purely meta-level interactions. e.g. a QoS Brokerage service in [27], a logger service in [25] and a distributed garbage collector service in [26].The development of suitable non-interference requirements allows us to reason about composition of multiple system services; these services have constraints that must be obeyed to maintain composability. We have identified three core activities:

- **Remote Creation:** - Recreation of services/data at a remote site. Remote creation can be used as the basis for designing algorithms for activities such as migration, replication and load balancing.
- **Distributed Snapshot:** Capturing information at multiple nodes/sites used as a basis for distributed garbage collection
- **Directory Services:** Interactions with a global repository. Directory services can be used to provide access control and implement group communication protocols.

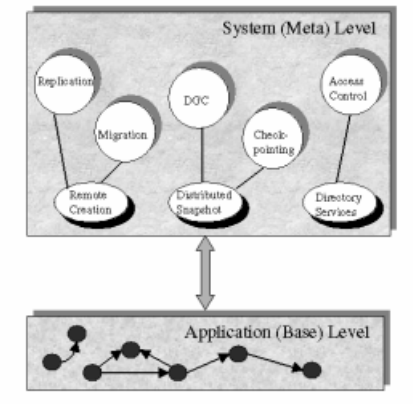

Fig. 1. Classification of core services

3 A Reflective Communication Framework

In this section, we describe a meta level communication framework that (a)provides run-time flexibility in the composition of communication protocols while ensuring their correctness and (b)ensures correctness of the resource management services (e.g. garbage collection, migration, etc.) in the presence of the flexible communication layer. To achieve this, the communication framework

must distinguish and handle different types of messages and communication protocols among objects (actors) in the system.

In order to provide correct composition of communication protocols in a transparent and scalable fashion, the TLAM model is extended with a reflective communication framework (RCF), which customizes the base level communication as follows (see Figure 2). Each base level actor has a meta level actor, called *messenger*, which serves as the customized and transparent mail queue for that base level actor. There is one *communication manager* in every node of the distributed system, which implements and controls the correct composition of communication protocols specified by messengers on that node. A messenger has four message queues:

- The *up* queue is used as the (base level) actor's send buffer.
- The *down* queue is used to deliver messages to the (base level) actor, serving as its customized mail queue.
- The *in* queue is used for interaction with the communication manager, requesting communication services of the incoming messages (to the base level actor).
- The *out* queue is used for interaction with the communication manager, requesting communication services for the outgoing messages (from the base level actor).

Hence, the *up* and *down* queues hold messages with no communication protocols attached to them (i.e. raw messages), while the *in* and *out* queues hold messages with communication protocols attached to them (i.e. processed messages). The *in* and *out* queues allow us to support timing QoS constraints through message priority, since messages may be reordered and served according to their priority.

The communication manager has a set of communication protocol actors, each of them implementing a particular communication protocol provided by the framework (e.g. reliable, in-order or security). This scheme allows us to abstract a core set of communication protocols and share it between the different messengers on a node, simplifying the synchronization and composition process. Furthermore, it encourages separation of concerns in the process of message transmission and reception. However, in purely reflective architectures, reasoning about the semantics of correct communication composition may be complicated; moreover, its implementation may be inefficient.

In order to maintain accurate semantics and provide an efficient implementation of the architecture, the communication manager implements a set of meta level entities, called communication message coordinators (or simply *message coordinators*).Every message requiring a communication protocol is assigned a message coordinator and at any instance, a message coordinator handles the composition of the communication protocols requested by a messenger for an individual message. The message coordinator assures the correct order of composition of required protocols and provides a coordination mechanism between the messenger and the protocols that provide it. This concept of reusable message coordinators is an efficient way to handle the service request of each messenger

without having to pay the bottleneck associated with the centralization of the services in the node communication manager and allows us to process concurrently multiple messages.

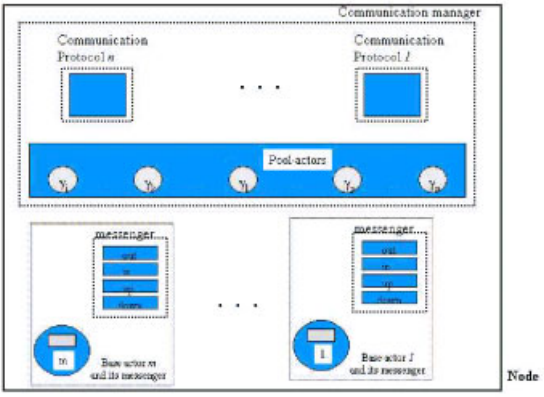

Fig. 2. The reflective communication framework model

In the RCF model, a communication protocol may be explicitly requested (at any instance) by the sender or implicitly specified by its messenger, in case the sender and the receiver previously agreed to use a particular communication protocol. In order to provide dynamic customization of communication protocols (potentially on a per message basis), the RCF model separates specification, composition and implementation of communication protocols.

The specification is handled by the messenger through a message service list (*msl*), which determines the set of communication protocols to be used in the communication of a specific message to its target. The communication protocols required by a communication may be explicitly specified or constructed by the messenger using target-actor information within the message. The messenger also assigns every outgoing message a unique message identifier (*msgid*), which is a unique sequence of values assigned to every message to be sent. The *msgid* is composed by concatenating the communication manager identifier (*cmid*), an application identifier (*appid*), a communication manager sequence number (*cms*) and a local time stamp (*ts*) defined by Lamport's happened-before relation [17]. The protocols themselves are implemented by independent communication protocol actors. This scheme allows us to add (*plug in*) or remove (*plug out*) communication protocols dynamically.

4 Protocol Interactions and Interference

Issues of correctness can be quite subtle and complex in dynamic asynchronous environments, where fluctuations in the communication environment may inter-

fere with the execution of communication protocols and middleware services. For a long time, customization of communication protocols has been seen as an isolated communication protocol composition problem and its interactions with other resource management (middleware) services underestimated. We have identified two common aspects that must be constrained in order to achieve safe flexibility while providing customization of communication services: Protocol interference and Protocol-Service interactions.

4.1 Protocol Interference

In order to support customization, the communication framework may need to alter the set of currently executing protocols. A newly chosen protocol set must be composed. This in turn implies that the composition order of the combined protocols be constrained in order to preserve their communication guarantees and obtain their combined benefits. Although the problem of determining the correct composition order of communication protocols has not been completely solved, several useful techniques have been developed. In particular, Horus [32] and Ensemble [22] provide a finite set of ready-to-use protocol stacks for group communication, each one of them implementing a specific set of properties. BAST [3] is more flexible and presents an extensible set of communication protocols to provide fault tolerance that can be combined at run-time. Unfortunately, protocol composition goes far beyond the mere layering order of the protocols due to subtle interaction properties that must be clearly spelled out.

Let us assume that we have an application server *svr* running in a private network and three mobile objects *a*,*b* and *c* that want to access time sensitive information from *svr* and communicate with each other in an ubiquitous reliable fashion. Since the communication environment may lack support for continuous flow of information, due to unpredictable network conditions, a timed delivery protocol *TD1* may be selected (transparent to the end user) to provide timed reliable delivery of messages within a certain period of time, using message counters and selected retransmission.

Assuming that *a*,*b* and *c* are moving towards an insecure environment, they may decide to add a secure communication protocol *S1* (in addition to the *TD1* protocol already provided), in order to ensure message integrity,secrecy and possibly authenticity.

As we previously described, mere layering *TD1* after *S1* or *S1* after *TD1* will not work, because *TD1* was not designed to deal with an insecure medium, which may corrupt or fake messages and the algorithm used in *S1* may increase the processing time required to encrypt/decrypt a message far beyond the message time span. Furthermore, most security protocols work under the assumption of a session based communication that may not hold in dynamic environments, where a frequent re-keying operation may need to be executed. Thus, messages may face temporary delays due to re-keying, contributing to further communication overhead. Under these circumstances, the framework should be able to select a more suitable security protocol *S2* (based on message digests, for in-

stance) in order to reduce the overhead and comply with the application timing requirement.

As can be observed, the time span (or deadline) is an interaction parameter that constrains communication protocol behaviors in order to achieve an application requirement (i.e. timed delivery). Usually, interaction parameters are entwined in the protocol functionality, for example, the message time-out values in a reliable delivery protocol.

We believe that for communication to be truly customizable, protocols must be encoded to separately specify (a) protocol functionality and (b) interaction properties and parameters. This will provide us the ability to determine what interaction parameters are affected when protocols are composed and allow us to modify these parameters independently if necessary. Furthermore, the chosen protocols may exhibit dependencies on other protocols or micro-protocols. Protocol dependency schemes have been researched and we have been able to integrate and leverage dependence relationships between protocols studied in other projects [4] [22].

This provides us with an initial sample set of communication protocols to uncover interaction properties that we are use in the development of a generic rule base. The rule base is used as an oracle using which the RCF determines which protocol implementation is more suitable to use for a particular application in terms of actual requirement coverage and efficiency.

Within the RCF, each protocol specification is enhanced with (i)a list of prerequisites (dependencies) *prlst*, (ii)a list of restrictions *rstlst* and (iii)a list of interaction parameters *iplst*. The *prlst* specifies requirements of the protocol in terms of other protocols; while the *rstlst* specifies restrictions on protocol composability (e.g. protocol *cp1* can not be executed with protocol *cp4*)[1]. The *iplst* specifies the parameters that might be adjusted in order to ensure safe composition of protocols.

Using the modular specifications described above, the message coordinator ensures safe protocol composition by coordinating and possibly constraining protocol execution order at the sender side as follows:

1. Extracts the *prlst* and *rstlst* of each protocol listed in the message service list (*msl*).
2. Creates a master list of *prlst* and *rstlst* by eliminating protocol redundancies in the individual *prlst* and *rstlst* lists.
3. Extracts from the message payload the application requirements.
4. Determines the required interaction parameters by using the information encoded in the *iplst* lists, the application requirements and the predefined rule base strategy.
5. Generates a composition order list *ordl* by cross referencing and using the information encoded in the *prlst* and *rstlst* master lists.
6. Verifies that the composition order obeys the interaction constraints and adjusts the required interaction parameters.

[1] The mechanisms for creation and formal verification of prerequisites and restrictions are beyond the scope of this paper

7. Serializes execution order if it is required to assure safe composition.
8. Coordinates communication protocol execution by enforcing execution constraints.

The above steps are mainly required at the sender side where a message is composed. Handling messages delivered at the receiver end is simple since the specific communication protocol set applied to the message payload must only be unwrapped.

4.2 Protocol-Service Interactions

In this section, we briefly illustrate protocol-service interaction issues that can arise when communication protocols (e.g. a reliable delivery protocol) and a middleware service (e.g. a migration service) interact. Here, we use the term *protocol* as an implementation of a particular semantic property (or property variant) in the communication environment, and *middleware service* as an activity on the end-point nodes to implement specific application requirements or tasks. Usually, middleware (resource management) services make assumptions about the underlying communication environment. If these assumptions are not satisfied by the communication subsystem, correctness violations, such as inconsistent states and incorrect execution semantics may occur. Let us consider the interaction between the migration service and the communication subsystem in dynamic environments. The migration service allows actor relocation for easier access, availability and load balancing. Let us further consider a specific implementation of the actor migration service that uses a 3 phase approach: (1)Phase C_0 determines the computation to be migrated by suspending the actor computation and noting its current description, (2)Phase C_1 creates a new remote actor with the current actor description in the desired node and (3)Phase C_3 establishes transparent access to the migrated actor by changing the original actor's behavior to a *forwarder* that redirects all the incoming messages to the new actor location. When an actor migrates every so often, a forwarder actor chain is built to redirect incoming messages The forwarder chain may contain cycles and does not provide information about the current actor location. The migration process can be enhanced to detect when an actor returns to a previously visited location and reconciliate the cycle to avoid unnecessary loops. This implies that the local migration manager on a node (which executes the cycle detection and reconciliation algorithm) has access to the entire history, i.e. forwarder chain of objects, of the migrating object.

Traversing this forwarder chain may take an unpredictable amount of time, especially if the actor is migrating during the traversal process.

Let us now assume that a reliable message delivery protocol is integrated into this framework. Since determining where the target actor of a message currently resides can possibly take an unpredictable amount of time, the timeout values encoded into the reliable delivery protocol may be meaningless. This can destroy message delivery guarantees even when the underlying transport layer is reliable. i.e. In the extreme case, if an actor is continuously migrated from one location to another, a message sent to it may be continuously forwarded to the previous

actor location, but never reaches the actor due to the frequent actor migration. In fact the problem is worse. It is complex for a sender to distinguish between a message that is in transit chasing its migrating destination and a message that is lost due to unreliable delivery.

Figure 3 illustrates this problem. Actors A and B on nodes N1 and N2 respectively exchange messages (m_1,m_2), then A decides to migrate to node N3(becoming actor$A$') and a forwarder chain (dashed line) is established between the actor's previous location (depicted as A) and its current location (depicted as A'). If B sent a message m_3 to A (at any point in time during or after the migration), m_3 will be forwarded to the new actor location (A') as soon as the migration has completed; meanwhile A' may decide to migrate again to node N4, now becoming A" and the forwarder chain is updated. Since m_3 must now traverse the forwarder chain, it is possible that the original timeout value may expire. B now sends another message m_4 (which could possibly be a retransmission of m3). Meanwhile, the actor continues to migrate asynchronously to node N5 (object A"') and then to node N6 (object A"") without any knowledge that messages m_3 and m_4 are chasing it. In fact, the problem get more complicated if B starts to migrate (which may be the case in actual applications). In this case we have acknowledgements chasing a migrating sender.

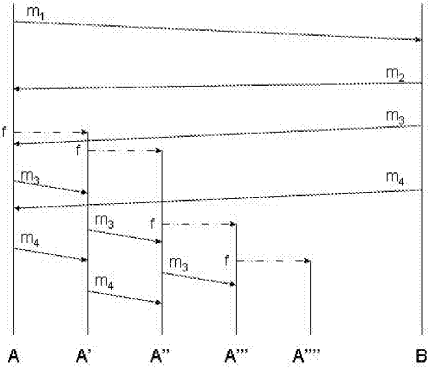

Fig. 3. Interaction diagram of the composition problem between a reliable delivery protocol and a migration service

A naive solution is the use of a flooding mechanism such as broadcast or multicast whereby every reliable message is broadcast to every node; the messaging layer can then determine if a current incarnation of the object resides on its node. Apart from being excessive in terms of messaging overhead, this solution requires that each object maintain the entire history of all possible previous names; it also requires that the messaging layer traverse the history chain of potentially every object on its node. We dismiss this solution due to the excessive overhead that will result in highly dynamic message-based environments with frequent object migrations.

A conservative solution is to constrain actor mobility in order to guarantee message delivery. A pessimistic approach may serialize the migration and reliable messaging sending processes (e.g by publishing well known intervals when objects may migrate) [2].

We propose a conservative (yet not fully pessimistic) solution where we constrain actor migration only during the time required to ensure message delivery to that specific object. This solution requires that some node in the forwarder chain realizes when a reliable message must be delivered to a local object and restrains the migration process until the desired messages are delivered. Assuming that we are able to reconciliate forwarder chains, the overhead of successful message delivery is upper bounded by $O(NP)$ where N is the total number of nodes in the system and P is maximum number of protocols that need to be applied to the message. That is, in the worst case, the actor may visit every node in the system before being notified that it cannot migrate until the chasing messages are delivered to it.

We now add a timed delivery requirement to this process. The above solution proposed can only ensure the worst case timing guarantees due to the fact that traversing a forwarder chain may take a significant amount of time, especially if the actor migrates frequently during the traversal process, the number of nodes in the network is huge and/or the protocol processing overheads are high.

We propose a more reasonable approach to enforcing timing constraints where we can upper bound message delivery overhead to be $OC + P$ where C is a constant representing the DS overhead. In this solution, a directory service always maintains the current location of the object (this is a feature already implemented in CompOSE|Q to ensure composability of middleware services). A timed (reliable) messaging protocol will now consult the DS to determine the current actor location. As a side effect, the DS entry is annotated with a *disable-migration* action. If a migration has already been initiated (but not completed) before the message from timed-delivery protocol reaches the DS, the existing (to become stale) location is returned to the timed-delivery protocol. The current migration is allowed to proceed asynchronously and succeed, a forwarder is established and a subsequent migration from the new location is stalled until the desired message has been acknowledged to be delivered. The timed reliable message will now need to traverse at most one forwarder link. The worst case time taken for reliable delivery is now the sum of a DS access time (constant) and the overhead of traversing one forwarder link. Note, however that this solution requires suitable enhancements to the reliable delivery protocol, the directory management modules and the migration service. The RCF modules can adaptively choose to traverse the forwarder chains or consult the DS based on the

[2] Note that placing strong constraints on actor behavior (ability to migrate in this case) can be overly restrictive and prevent adaptation to system and application criteria. In fact, they may be used by an arbitrary (malicious) actor to produce byzantine failures.

timeliness requirements of the reliable message and some information on the frequency of migration of the object under consideration. For further details on the correctness reasoning of the above protocol-service composition, refer to[34].

5 Implementation and Performance

In order to provide an implementation environment of the RCF, we constrain the *Java* programming language to achieve actor semantics [28][33][6]. Since a messenger may have multiple communication request in its queues at any point in the execution, our objective is to exploit maximal concurrency in the processing of messages without violating actor semantics. This in term requires that access to the multiple messenger queues be appropriately synchronized.

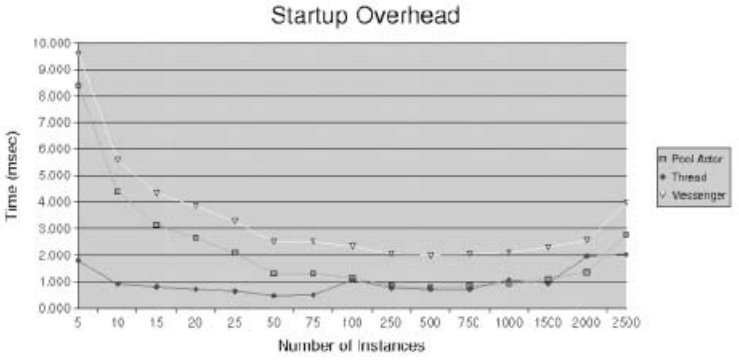

Fig. 4. Thread, Pool Actor and Messenger creation overhead

We adopt a *pipelined* approach to execute multiple communication operations within a messenger. we also provide for the activation of external (new or pre-existing) actors within messages, thereby supporting a continuation pattern style (CPS) implementation [5]. To achieve this, we extend the *Active Object pattern* appropriately [31] and instantiate CPS execution via the *command pattern* [8]. The main advantage of this approach is that the individual operations on a messenger need not be strictly sequential. In fact, the only synchronization necessary is in the messenger's queues, since more than one message coordinator, the runtime transport layer and the messenger itself might try to dequeue a message from the same queue or from an empty queue or enqueue a message into the same queue. To facilitate this, we developed a blocking queue mechanism based on a linked list implementation (see Table 2), in order to ensure correct execution. Although the overhead in Java for synchronized access to the messenger queues is quite expensive; we have observed that it is a fixed cost that can be amortized by pipelining as the number of messages in the system increases. This amortization is bounded by the available heap space allocated to the runtime environment.

Table 1. Creation overhead of the RCF: The Communication manager has three communication protocols and a pool size of ten message coordinators. The Messenger creation is slightly more expensive because it has 4 queues that must be initialized

Creation Overhead	*μsec*
Communication Manager Pool (10) (9960) Protocols(3) (2990)	12950
Message Coordinator	996
Messenger	1203

The platform used to measure the performance of the RCF model are Sun Ultra5 workstations (333Mhz UltraSPARC IIi with 256KB external cache and 128 MB RAM) running Solaris 2.7 connected via a 100Mb/s Ethernet link. The performance metric results presented below were obtained without timing *inlined JVM* internal calls using JDK-1.2.2 with green threads:

1. **RCF Startup Overhead:** The amount of time required for the communication manager startup is dependent of the number of protocols initially implemented and the size of the (message coordinator) pool specified. Initially, we decided to model 3 protocols and a pool size of 10 message coordinators (Table 1). Although expensive at startup in terms of system resources, the pool really improves the performance at run-time because message coordinators are awakened when needed instead of being created at run-time, which can be quite expensive as Figure 4 shows. Note that the high cost paid by the messenger and message coordinators at the beginning is due to the initialization of the blocking queues, four for every messenger and one for each message coordinator. Unfortunately, messenger creation happens at run-time, which may decrease system performance. An alternative solution would be to create a pool of messengers at startup and link them at run-time.
2. **Messenger Management Overhead:** During messenger creation, 4 blocking queues are created and initialized. Queue initialization is expensive due to the linked list implementation and the spin lock required to avoid race conditions in message dequeue operations. Table 2 shows the blocking queue overhead in terms of its common operations.

Table 2. Blocking Queue Operations:

Blocking Queue Operations	*μsec*
Initialization	1136
Enqueue	308
Dequeue	325
Flush	160

Since every messenger has 4 blocking queues that can hold either raw messages (*i.e.*, messages without protocol services attached to it) or processed messages, and it must be able to inspect the contents of enqueued messages without dequeueing them, messenger management overhead becomes critical (Table 3).

Table 3. Message Operations:

Message Operations	μsec
Message Inspection	223
Raw Message Enqueue	489
Raw Message Dequeue	466
Processed Message Enqueue	1710
Processed Message Dequeue	1801

3. **Total Message Overhead:** In order to measure the end-to-end message overhead, we use *CompOSE*$|Q$ as our underlying run-time environment, which consists of a set of runtime kernels that run on individual nodes of the distributed system and middleware components that provide the required distributed services (*i.e.* object migration, directory services or distributed garbage collection). The node runtime kernel is a substrate on which actors execute. The principal components of a node runtime kernel are:
   a) A *node manager* actor, which manages and coordinates various components in a node.
   b) A *node info manager* actors, which manages information needed by local actors and interfaces with a global (logically centralized) directory service.
   c) A transport subsystem that handles messages between actors. It provides a framework for sending the the outgoing messages to the appropriate node (routing) and resolving incoming messages to their appropriate actor queues (message resolution).

   As we can see in Table 4, the end-to-end message overhead is measured by categorizing the different overheads involved in the process. First we measured the message transmission and reception overheads of the underlying transport layer. Then, we integrated our model into the framework and measured the time needed to send a raw message and a message tunneled through the RCF. Finally, we send processed messages using a reliable secure customization.
   a) **Raw Message Overhead:** During system startup a *communication manager* is instantiated in each node of the system. When an actor is created and protocol composition services are not desired or required, a corresponding *messenger* is not created; the actor sends and receives raw messages using the underlying message transport layer.
   b) **RCF Tunneling Overhead:** When flexible communication is required or desired, an independent *messenger* is created for every base level actor

and the entire RCF functionality is invoked. Since an actor can potentially communicate differently (using different protocols or not using any protocol) with each of its acquaintances at any time, the overhead of the RCF model must be minimized in the case of communications with no protocols attached. In this scenario, we tunnel raw messages through the actor's *messenger* directly to the underlying message transport layer.

c) **Processed Message Overhead:**Because most of the protocols need to add some information to the message (in the form of headers), its execution contributes to some processing overhead, which increases the total time by one order of magnitude. Reducing the overhead of protocol layering is a difficult task; fortunately, this cost is almost neglected in terms of remote communication due to the roundtrip delay, which is estimated to be between 20-41 milliseconds. Our actual implementation is suboptimal since a naive implementation of the protocol results in the protocol overhead being around 0.622 milliseconds. Possible optimizations may be obtained if we replicate stateless protocols[3], and improve current protocol implementations.

Table 4. Message Transmission (Tx) and Reception (Rx) Overhead:

Message Overhead	Tx (μsec)	Rx (μsec)
Raw Message	181	141
RCF Tunneling	670	607
Processed Message	1891	1942

6 Related Work and Concluding Remarks

Traditional reflective languages aim at providing a customizable execution of concurrent systems. Former approaches to composition of communication services [13] [20][7][21] assume point-to-point communication and do not impose formal restrictions on the structure and semantics of basic communication primitives. Basically, they use the onion skin model as a conceptual foundation for separation of concerns. [15] presents a formal specification of communication services in rewriting theory. However, they do not deal with dynamic installation of communication services and they synchronize each object with its meta object. A Similar approach is taken in [18], where they use the concept of *weaving* to deal with the problems posed by composition of distributed (point-to-point) communication services. Similarly, [14] provides a meta level architecture at the language level to support a limited fault tolerant system mechanisms, in particular replication.

[3] Stateless protocols do not need to keep softstate information about every message sent or received

Commercially available distributed middleware infrastructures have incorporated the notion of reflection in order to provide the desired level of configurability and openness in a controlled manner [9][2]. However, they do not provide semantics that accurately describe the composition of communication services and they do not deal with the implication of composing services and communication protocols. *e.g.* The pluggable protocol framework [1] addresses the lack of support for multiple inter-ORB protocols and deals with integration and use of multiple ORB messaging and transport protocols, not with the composition of the protocols itself. DynamicTAO [10] explore ways to make the various components of an ORB dynamically configurable as well as componentizing them to achieve minimal footprint for small applications [19].

In recent years, several group communication systems (GCS) have been developed to provide additional flexibility of communication services by allowing for dynamic composition of communication protocols [16] [32] [22] [23]. Specification and formal characterization of group communication services has been developed in the context of *I/O Automata* [29]. Often, the communication mechanisms are built into the architecture; dynamic installation and revocation of communication protocols on-the-fly to deal with such changes are cumbersome and error-prone.

The designed communication framework provides flexibility in composition and dynamic installation of communication protocols, and provides the first step in an effort to provide a cost-effective communication framework capable of addressing adaptive environments with evolving policies. However, the integration of the RCF with reflective middleware services requires further investigation; the overall middleware environment must allow the service and protocol implementations to adapt themselves based on existing system conditions and application requirements.

Future work includes the development of an extensive suite of interaction properties for various communication protocols that implement application requirements and the creation of a generic rule base strategy to manage interprotocol interactions.

References

[1] Alexander Arulanthu, Carlos O'Ryan, Douglas C Schmidt, Michael Kircher and Jeff Parsons. The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging. In *Proceedings of the IFIP/ACM Middleware 2000*, 2000.

[2] Ashish Singhai, Aamod Sane and Roy Campbel. Reflective ORBs: supporting robust, time-critical distribution. In *Proceedings of the European Conference on Object-Oriented Programing*, 1997.

[3] Benoit Garbinato and Rachid Guerraoui. Using the Strategy Design Pattern to Compose Reliable Distributed Protocols. In *Usenix Conference on Object-Oriented Technologies and Systems*, 1997.

[4] Benoit Garbinato and Rachid Guerraoui. Flexible Protocol Composition in BAST. In *IEEE International Conference on Distributed Computing Systems*, 1998.

[5] Carlos Varela and Gul Agha. Linguistic Support for Actors, First-Class Token-Passing Continuations and Join Continuations. In *Midwest Society for Programming Languages and Systems Workshop*, 1999.
[6] Carlos Varela and Gul Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Intriguing Technology Track*, 2001.
[7] Daniel Sturman. *Modular Specification of Interaction of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaing, 1996.
[8] Erich Gamma, Richard Helm, Ralp Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
[9] Fabio Costa, Gordon Blairand Geoff Coulson. Experiments with Reflective Middleware. Technical Report MPG-98-11, Lancaster University, 1998.
[10] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz C Magalhães and Roy Campbell. Monitoring and Security and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM Middleware 2000*, 2000.
[11] Gul Agha. *Actors: A model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
[12] Gul Agha and Ian A Mason and Scott F Smith and Carolyn Talcott. A Foundation for Actor Computation. *Functional Programming*, 1993.
[13] Gul Agha, Svend Frolund, Rajendra Panwar and Daniel Sturman. A Linguistic Framework for Dynamic Composition of Dependability Protocols. In *Dependable Computing and Fault Tolerant Systems*, 1993.
[14] Jean-Charles Fabre and Tanguy Perennou. A Metaobject Architecture for Fault-tolerant Distributed Systems: The FRIENDS Approach. In *IEEE Transactions on Computers, (47):78-95*, 1998.
[15] Jose Meseguer, Carolyn Talcott and Denker G. Rewriting Semantics of Meta-Objects and Composable Distributed Services. SRI International, 1999.
[16] Larry Peterson, Norm Hutchinson, Sean O'Malley and Mark Abbot. RPC in the x-Kernel: Evaluating New Design Techniques. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, 1993.
[17] Leslie Lamport. Time and clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.
[18] Lynne Blair and Gordon Blair. The Impact of Aspect-Oriented Programming on Formal Methods. In *Proceedings of the European Conference on Object-Oriented Programing*, 1998.
[19] Manuel Roman, Dennis Mickunas, Fabio Kon and Roy Campbell. LegORB and Ubiquitous CORBA. In *Proceedings of the IFIP/ACM Workshop on Reflective Middleware 2000*, 2000.
[20] Mark Astley and Gul Agha. Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management. In *6th International Symposium on the foundation of Software Engineering*, 1998.
[21] Mark Astley, Daniel Sturman and Gul Agha. Customizable Middleware for Distributed Software. Communication of the ACM, 2000.
[22] Mark Garland Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998. Department of Computer Science.
[23] Matti A. Hiltunen, Vijaykumar Immanuel and Richard D. Schlichting. Supporting Customized Failure Models for Distributed Software. In *Distributed System Engineering, (6):103-111*, 1999.

[24] Nalini Venkatasubramanian. ComPOSE—Q - A QoS-enabled Customizable Middleware Framework for Distributed Computing. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 1999.

[25] Nalini Venkatasubramanian and Carolyn Talcott. A Semantic Framework for Modeling and Reasoning about Reflective Middleware. In *IEEE Distributed Systems Online, 2(6)*, 2001.

[26] Nalini Venkatasubramanian and Carolyn Talcott. A Formal Correctness Proof for the Hierarchical Distributed Garbage Collection Algorithm. Technical Report TR-Stanford, Stanford University, 2002.

[27] Nalini Venkatasubramanian, Carolyn Talcott and Gul Agha. A Formal Model for Reasoning about Adaptive QoS-Enabled Middleware. In *Formal Methods Europe (FME 2001)*, 2001.

[28] Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Sebastian Gutierrez-Nolasco and Jehan Wickramasuriya. Design and Implementation of a Composable Reflective Middleware Framework. In *International Conference on Distributed Computer Systems*, 2001.

[29] Nancy Lynch and M R Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219-246, 1989.

[30] Priya Narasimhan, Luise E Moser and P M Melliar-Smith. Using Interceptors to Enhance CORBA. *IEEE Computer Magazine*, 1999.

[31] R.G. Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Pattern Languages of Program Design*, 1996.

[32] Robbert van Renesse, Kenneth Birman and Silvano Maffeis. Horus: A Flexible Group Communication System. *Communication of the ACM*, 39(4):76-83, 1996.

[33] Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian. Design Patterns for Safe Reflective Middleware. In *OOPSLA, Workshop Towards Patterns and Pattern Languages for Object-Oriented Distributed Real-Time and Embedded Systems*, 2001.

[34] Sebastian Gutierrez-Nolasco and Nalini Venkatasubramanian. Reliable Communication with Mobile Objects in Faulty Environments. Technical Report TR-DSM-01-04, University of California, Irvine, 2001.