A Customized Communication Subsystem for FT-Linda*

Dorgival O. Guedes† *David E. Bakken*‡ *Nina T. Bhatti*
Matti A. Hiltunen *Richard D. Schlichting*

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

E-mail: {dorgival,bakken,nina,matti,rick}@cs.arizona.edu

Abstract

Distributed fault-tolerant systems usually impose much stronger requirements on the underlying communication protocols than do applications developed without fault-tolerance in mind. That is true, for example, of applications composed of processes replicated on multiple hosts, where all replicas must keep the same view of the state of the communication. This paper describes how the communication substrate for a specific application with strong communication requirements was developed. The application, the runtime system for a fault-tolerant version of the Linda language called FT-Linda, requires a communication substrate capable of providing ordered atomic multicast, failure detection and membership services. The implementation relies on a new framework for the composition of event-driven micro-protocols that is used with the x-kernel.

Resumo

Sistemas distribuídos tolerantes a falhas usualmente impõem maiores exigências sobre os protocolos de comunicação utilizados do que aplicações desenvolvidas sem o objetivo de se prover tolerância a falhas. Tal fato ocorre por exemplo com aplicações desenvolvidas replicando-se um processo em várias máquinas, onde todas as réplicas devem manter a mesma visão do estado do sistema. Este artigo descreve o desenvolvimento de um protocolo uma aplicação específica, o ambiente de execução de FT-Linda, uma versão tolerante a falhas da linguagem de coordenação Linda. O sistema exige um protocolo de comunicação capaz de prover detecção de falhas, identificação de participantes e um serviço de "multicast" confiável. A implementação se basea em um novo ambiente para composição de micro-protocolos acionados por eventos que é utilizado em combinação com o x-kernel.

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

†Sponsored by Conselho Nacional de Pesquisa (CNPq), Brazil, Process no. 200861/93-0

‡Current address: Distributed Systems Department, BBN Systems and Technologies, 10 Moulton Street MS 6/3D, Cambridge, MA 02138 USA

1 Introduction

Distributed systems are used nowadays in applications that require dependable service, which poses difficult problems for the implementor of such systems. Although the general issue of fault-tolerant distributed systems is undoubtedly a hard one, techniques have been developed that can be used by developers to implement those systems according to well-known paradigms. Among these techniques, communication protocols that provide elaborate services like failure detection, membership management and atomic multicast are some of the important building blocks available [19, 7, 11].

Due to the strong requirements imposed on fault-tolerant systems, such protocols are usually extremely complex and constrained to provide only a single set of rigidly defined semantics, which makes them very difficult to construct and use. A new approach for addressing these problems has been developed in which properties can be implemented as separate *micro-protocols* and then configured together to construct a higher-level *composite protocol* that provides a customized service. Each part of the service can be implemented and tuned separately, while the interaction among the parts can be explicitly defined.

This approach makes it easier to develop the services, since each micro-protocol can be implemented as a separate logical unit, and each unit can be adapted to a certain given application. Altering the behavior of the protocol to adjust to changes in the required service is easily done by replacing or adjusting the micro-protocols responsible for a certain feature. The modularity also makes it much simpler to develop and debug the composite protocol, since the interfaces among the parts are well defined.

This paper describes a case study in which a customized service is constructed using this approach for FT-Linda, a version of the Linda coordination language designed for writing fault-tolerant parallel programs [2]. The service, an atomic ordered multicast protocol that is used as the communication substrate for the language runtime system, has proven to be difficult to construct in practice. This has been demonstrated, for example, by earlier experience with Consul, the protocol suite previously used by FT-Linda [14]. When compared to the original design done with Consul, the system constructed using micro-protocols is simpler and better tuned to the special characteristics of the FT-Linda runtime system. It also allows various different aspects of the service to be altered in order to determine the best implementation for the given system.

The remainder of this paper is organized as follows. Section 2 describes FT-Linda, its motivation, the new syntax and semantics, and the communication needs of the runtime system. Once those requirements have been defined, Section 3 describes the event-driven protocol composition model and the structure of the intended composite protocol. Section 4 then provides the details of the implementation, with the important data structures and the outline of micro-protocols. Finally, Section 5 offers some concluding remarks and discusses possible future work.

It should be noted that the description of FT-Linda presented here was developed as a short introduction to provide just the background necessary for the analysis of the underlying communication substrate, the real focus of this paper. A detailed description of the semantics and implementation can be found in [1].

2 FT-Linda[1]

Linda basic concepts. Linda is a language for parallel programming based on *tuple spaces* (TS), a communication abstraction defined as a bag that can hold data elements called tuples. These tuples are data aggregates that have a logical name and zero or more values. The tuple spaces are, in essence, a specialized virtual shared memory, in which the sharing of information is guaranteed by the runtime system.

Processes can use TS to communicate and synchronize with other processes by manipulating tuples. Such manipulation is done through a set of basic operations to deposit and withdraw tuples from a TS. These operations are **out**, which deposits a tuple, and **in**, which withdraws a tuple with specified characteristics if available and blocks otherwise. Other operations are defined, like **rd**, **inp** and **rdp**, but these are basically adaptations of **in** and **out**. Figure 1 shows how Linda can be used to implement a worker process under the bag-of-tasks paradigm [5].

```
process worker
   while true do
      in("work",?subtask_args)
      calc(subtask_args, var result_args)
      for (all new subtasks created by this subtask)
         out("work", new_subtask_args)
      out("result",result_args)
   end while
end proc
```

Figure 1: Bag-of-Tasks Worker

Problems with failures. The standard definition of the language and its operations does not address the effects of processor failures, however. There are essentially two deficiencies in the model that make it susceptible to failures:

Lack of tuple stability: The language does not define how the runtime system must store tuples in order to create the illusion of shared memory. Many current implementations use some kind of signature to partition tuples among the participating hosts. On these systems, the failure of a host may cause the loss of an unpredictable subset of tuples in the TS [8].

Lack of sufficient atomicity: Only the basic operations are defined to be atomic in Linda, what means intermediate states during the execution of a series of operations can be seen by other processes. If the processor on which the operations are being executed fails before the completion of the task, the TS may be left in an indeterminate state. For example, in figure 1, no result would be output to TS if the processor hosting the worker fails while executing `calc`.

FT-Linda extends the original Linda model with stable tuple spaces and atomic execution of sequences of operations to provide improved support for building fault-tolerant applications. The model assumes that processors suffer only fail-stop failures [18], where the runtime system provide failure notification by depositing a distinguished *failure tuple* into TS. Currently it assumes that processors remain failed for the duration of the computation and are not reintegrated back into the system.

[1] A detailed description of the language and its implementation can be found in [1]

2.1 Syntax and semantics

To address the deficiencies mentioned above, FT-Linda includes provisions for defining *stable TSs* and a new syntax that allows a series of TS operations to be defined as atomic.

Stable tuple spaces. The original Linda language defined only one globally visible TS that is shared by all applications. After that, many different studies have suggested various features to allow multiple TSs to be defined under the control of the application [8, 6]. That feature is incorporated in FT-Linda, and is further extended with attributes.

Tuple spaces may be assigned special attributes that define how they behave in the presence of failures, among other issues. Currently these attributes are *resilience* and *scope*. Resilience specifies the behavior of the TS in the presence of failures, and can be set as *stable* or *volatile*: The first guarantees that the TS will survive processor failures, while the second makes no such guarantee. The scope attribute indicates which processes can access the TS, and can be *shared* (all processes may access it), or *private* (only one process has access).

Stability is achieved by replicating tuples on multiple machines, which is also used to implement a shared TS since processes on any host may require access to its tuples. On the other hand, volatile private TSs may be implemented locally to the owning process, providing a faster access to a local work area where temporary results may be stored. As described below, new atomic operations are provided to move the contents of a local TS to a shared/stable one when a complex series of operations is finished, providing one of the ways to ensure atomicity.

One more semantic extension provided by FT-Linda is that tuple spaces preserve the order of insertion. That is, tuples in a given TS are always ordered in this way by the runtime system, a guarantee that can be exploited to good effect by many applications.

Failure detection and notification When the system detects a host failure, it automatically creates a *failure* tuple in a shared stable TS available to all processes in the application. Each application must define a process responsible for watching for those tuples and starting the adequate recovery procedure.

Atomic guarded statements (AGS). An AGS is a new construct in the language that allows a programmer to specify that a group of tuple space operations be executed atomically, potentially after blocking to wait for a condition to hold. This provides all-or-none execution semantics despite failures or concurrent access to TS by other processes.

The simplest case of the AGS is $< guard \Longrightarrow body >$ where the angle brackets denote atomic execution. The *guard* can be any blocking Linda operation or **true**, and the *body* is a series of **in**, **rd**, **out**, **move** or **copy** operations, or a null body denoted by **skip**.

A process executing an AGS is blocked until the guard succeeds, at which point the guard and body are executed as a single atomic step. Only the guard expression can block: if the body has any **in** or **rd** operation that would block, an error is reported.

A disjunctive case is also defined in which more than one guard/tuple pair can be specified. A process executing such a statement blocks until at least one of the guards succeeds, at which point one of the pairs is chosen to be executed atomically.

Atomic tuple transfer. FT-Linda provides primitives that allow tuples to be moved (**move**) or copied (**copy**) atomically between TSs.

An example. As mentioned above, these new FT-Linda features can be used to guarantee that other processes see a given task as atomic. For example, consider the bag-of-tasks application. After withdrawing a *work* tuple, a worker can generate a variable number of new tasks by creating new *work* tuples, followed by creation of a *result* tuple. In order to make these actions atomic, the description of the task can be removed from the TS only in an operation that atomically deposits any new *work* tuples and the *result* tuple. In this way, a failure while the worker is computing the result does not cause the TS to be in an inconsistent state. A possible solution is shown in figure 2, where the worker uses a local TS to create the new tasks and result, and then moves all the tuples atomically to the shared TS when done. Due to the atomic nature of the move operation, either all tuples appear in the TS at the same time, or, if the worker's processor fails before the last AGS can be executed, none do.

```
process worker()
    TSscratch := ts_create(volatile, private, my_lpid() )
    while true do
        ⟨ in (TSmain, "subtask", ?subtask_args) ⟹
            out (TSmain, "in_progress", my_hostid, subtask_args) ⟩
        calc (subtask_args, var res_args)
        for (all new subtasks created by this subtask)
            out(TSscratch, "subtask", new_subtask_args)
        out(TSscratch, "result",res_args)
        ⟨ in(TSmain, "in_progress", my_hostid, subtask_args) ⟹
            move (TSscratch, TSmain) ⟩
    end while
end worker
```

Figure 2: Dynamic Fault-Tolerant Bag-of-Tasks

Another important aspect of this implementation is the replacement of the initial *work* tuple by an *in_progress* tuple. This *in_progress* tuple can be used by a *monitor process* to recreate the *work* tuple in case the worker fails before completion. Such a monitor would execute in a loop waiting for *failure* tuples created by the runtime system when a host is detected to have failed. When such a tuple is found, it provides the *id* of the failed host, so the monitor can remove any *in_progress* tuples from that host and replace them by the original *work* tuples. In this way, all interrupted tasks can be processed later by other workers.

More details on this and other examples can be found in [1, 2].

2.2 FT-Linda implementation

Given the semantics just described, the challenge is to provide a reasonable implementation of atomic execution and stable TSs. The choices for the second range from using hardware assistance to approximate the failure-free behavior of stable storage to replicating the values (tuples) in volatile memory across multiple processors, so that failure of some of them can be tolerated without loss of information. Since the situation at hand also requires that tuples be shared among different processors, replication is a better choice.

To implement replicated TSs we use the *replicated state machine approach (SMA)* [20]. In this technique, a fault-tolerant application is implemented as a state machine that contains state variables and makes modifications in response to commands from other state machines or external sources. Resilience to failures is achieved by replicating the state machine on multiple independent processors and using an *ordered atomic multicast* to deliver commands to all replicas reliably and in the same order. This ordered delivery guarantees that all replicas execute all commands in the same order. Provided that the commands are deterministic and executed atomically in relation to each other, state variables of all replicas are kept consistent. The SMA is the basis for a large number of fault-tolerant systems [4, 14, 17].

Achieving atomicity can be done by exploiting the characteristics of the SMA, since each command is executed by a state machine atomically and the underlying atomic multicast guarantees that all commands are executed in the same order. Hence, a simple scheme is to treat the entire sequence of TS operations in an AGS as a single command to the state machines. Operations are disseminated in a single multicast message, which is executed by all state machines as dictated by the ordering realized by the atomic multicast. Since all replicas receive and execute all commands in the same order, they keep the same view of the TSs.

The implementation of FT-Linda is divided into four major parts:

- A *pre-compiler*, which translates a C program with FT-Linda constructs into C generated code (GC).
- The *FT-Linda library* (FTlib), which is responsible for handling the communication of the GC with the rest of the system and for implementing local TSs, improving their performance.
- The *TS state machine* (TSSM), which handles replicated copies of the TSs on each host using the SMA.
- The communication substrate, which provides the atomic multicast service to the TS state machines in the system.

The relation among the components is shown in figure 3. Requests from the GC are passed by the TSSM directly to the substrate to be distributed to all replicas. The state machines execute TS operations based on the order provided by substrate, with the replica on the same host as the process that initiated the request being responsible for handling any return values.

In our design, the TS state machines and the communication substrate are implemented as protocols in the x-kernel, a system for composing network protocols [12]. The state machine can be implemented easily using the abstractions provided by the x-kernel; specifically, the arrival of a message causes the proper operations in the state machine to be invoked. The reader is refered to [1] for a detailed description of this implementation. The communication substrate, on the other hand, is much more difficult to implement given that it incorporates a great variety of tasks on which the entire system depends.

One approach to implementing the communication substrate is to use an available atomic multicast protocol, like Consul [14], and just provide an interface between it and the TSSM. Another approach is to develop a new protocol specially tailored to the needs of the system. This approach has the advantages of allowing a better customization of the system, and if done in a modular way, can be altered and extended as necessary for

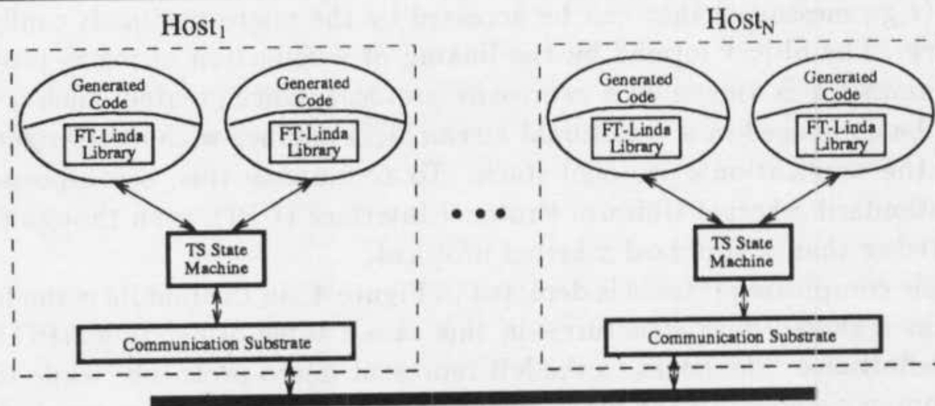

Figure 3: Runtime Structure

future extensions to the language. The remainder of this paper will show how we have explored this approach.

3 The communication subsystem

A protocol providing an atomic multicast service can be abstractly decomposed in a set of related tasks. The implementation can then be based on such a decomposition. As the experience with Consul shows, this can make the development much easier to understand, although the interaction among the parts can be difficult to model using the x-kernel [15]. It was with these considerations in mind that a new approach for implementing fault-tolerant services using micro-protocols and event-driven execution was developed.

3.1 Event-driven protocol composition[2]

A communication substrate with the required properties is realized using a model for composing fine-grained software modules [9] and its associated x-kernel based implementation platform [3]. The basic building block of this model is a collection of *micro-protocols*, each of which implements a well-defined property. A micro-protocol, in turn, is structured as a collection of *event handlers*, which are procedure-like segments of code that are invoked when an *event* occurs. Events can be either user or system defined, and are used to signify changes of state potentially of interest to the micro-protocol. For example, a commonly-used event for building network protocols like RPC is "message arrival." When an event is detected, all event handlers registered for that event are invoked; events can also be generated explicitly by micro-protocols, with the same effect. The invocation of event handlers due to the occurrence of a single event can be *sequential*—performed sequentially using one thread of control, or *concurrent*—performed concurrently with each event handler given its own thread of control. The invocation itself can be *blocking*, where the invoker waits until all the event handlers registered for the event have finished execution, or *non-blocking*, where the invoker continues execution without waiting.

Event registration, detection, and invocation are implemented by a standard runtime or *framework* that is linked with the micro-protocols. The framework also supports

[2] Text for this sub-section appeared previously in [10]

shared data (e.g., messages) that can be accessed by the micro-protocols configured into the framework. The object formed by the linking of a collection of micro-protocols and associated framework is known as a *composite protocol.* Once created, such a composite protocol can be composed in a traditional hierarchical manner with other x-kernel protocols to form the application's protocol stack. To accomplish this, a composite protocol exports the standard x-kernel Uniform Protocol Interface (UPI), even though its internal structure is richer than a standard x-kernel protocol.

An example composite protocol is depicted in Figure 4. In the middle is the framework, which contains a shared data structure—in this case a table of pending RPC calls—and some event definitions. The boxes to the left represent micro-protocols, while to the right are some common events with the list of micro-protocols that are to be invoked when the event occurs.

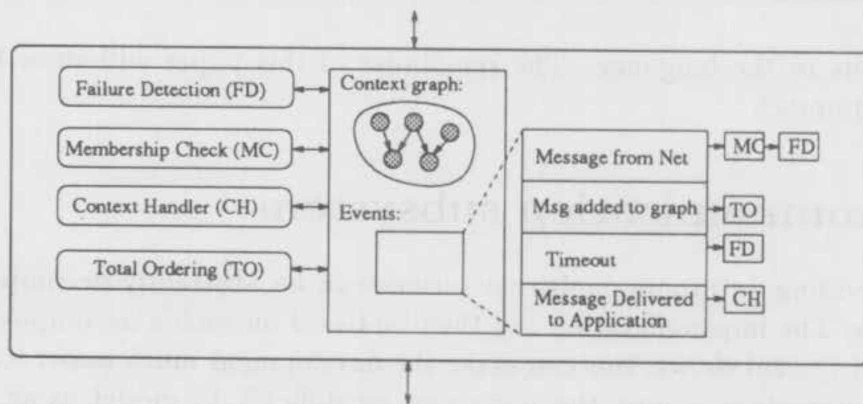

Figure 4: A composite protocol

The following operations are provided to micro-protocols by the framework for dealing with events.

- **register**(*event_name*,*event_handler_name*, *priority*), which is used to request that the framework invoke handler *event_handler_name* when *event_name* occurs. If the event is sequential, the event handlers registered for the event are executed in priority order based on the *priority* value each supplied when they registered. If omitted, the value defaults to the lowest priority.
- **trigger**(*event_name*,*arguments*), which is used to notify the framework that event *event_name* has occurred. The framework will then execute all the event handlers registered for this event, passing *arguments* in the invocation.
- **deregister**(*event_name*,*event_handler_name*), which is used to reverse the registration process.
- **cancel_event**(), which is used to notify the framework that the current event is to be cancelled, i.e., the remaining event handlers registered for this event need not be executed. This operation is mostly useful for sequential events.

The model also has a provision for events triggered by the passage of time. To request this, a micro-protocol uses the register procedure with TIMEOUT as the event name and specifies the time interval as the priority parameter. With the exception of the TIMEOUT

event, event handlers remain registered for their event until explicitly deregistered, so that each may be invoked any number of times. Event handlers registered for the TIMEOUT event are executed only once after the timeout period has expired.

3.2 Design overview

A first version of FT-Linda used Consul as the communication substrate. Based on that, we knew that the services it provides would be enough for the needs of the runtime system, but we wanted to experiment with a more custom-tailored protocol. From the early observations with the system, some facts were verified:

- In order to guarantee the proper operation of all non-failed copies of the TSSM, atomic ordered multicast is mandatory.
- The TSSMs are responsible for generating failure tuples in case a host is found to have failed, so failure detection and notification is necessary.
- The current FT-Linda implementation does not allow the re-integration of failed hosts, so the membership protocol can be a simple one: Hosts can only leave the group due to a failure.
- Once a message is passed to the TSSM, no new requests concerning that message will ever happen, which makes message handling much easier than in Consul.

Given these observations, then, the tasks that need to be performed by the communication substrate can be described as follows:

Validity check: Messages that are not from members of the group must be filtered out.

Monitoring: In order to implement failure detection, each host with an instance of the protocol must be able to detect when another host has been silent for longer than a fixed interval.

Liveness: To make it easier for the monitoring task to take decisions about the silence of a host, a micro-protocol must be responsible for ensuring that each host sends some message from time to time, even if the application does not.

Membership check: Information about a host that has not been transmitting for some time must be exchanged with other hosts before some agreement is reached about the state of a host suspected to be down.

Context graph handling: As in Psync [16], the ordering of messages is achieved by first determining their causal ordering [13]. That information is kept as a *context graph* representing the causal dependencies between messages.

Reliability: The substrate may have to keep track of lost messages and be able to request their retransmission.

Ordering: The causal ordering represented by the context graph is not enough to guarantee that all messages will be delivered in the same order to all hosts, so this causal order must be be extended to a total order.

Stability: As discussed below, stability is a key concept when defining a stronger ordering among messages, since it determines which messages have already been received by the application at all hosts.

The exact implementation of these abstractions is discussed in the next section.

4 Implementation

The previous section detailed the communication service required by the FT-Linda runtime system. This section describes how such a service can be implemented using the event-driven composition framework. When working with the framework, we can implement each of the tasks previously identified as a separate micro-protocol and use events to define the relations among them. The algorithms involved are directly derived from those developed for Psync and Consul, and are described only briefly. The reader is directed to [14] and [16] for a more detailed discussion.

4.1 Data structures

The implementation is built around three main data structures, usually handled by specific micro-protocols, but of importance for the correct operation of the entire composite protocol. The first is the context graph, which is used to keep track of the relative positions of the messages based on causality. It is the main structure to hold messages within the substrate before their position in the ordering can be determined.

The second data structure is used to maintain the group of hosts currently taking part in the FT-Linda computation. As might be expected, this is handled mostly by the micro-protocol responsible for keeping track of group membership as execution proceeds.

Finally, we have the message data structure. It contains the message data itself, which is usually opaque to the framework, and the message attributes, which represent information related to the message used by the micro-protocols. These attributes are transmitted between instances of the protocol as part of the message header, and are the following in this case:

sender_id: The identification of the sender of the message.

msg_num: The number assigned to a message by the sender.

msg_direction: Needed to differentiate between messages traveling up and down the protocol graph.

msg_type: Whether the message contains data from the application (TSSM), or liveness and membership information, among others.

predecessors: The messages that precede a given message in the context graph. They are defined as the most recent messages from each host in the local graph at the time the TSSM passes a message down to be transmitted.

successors: Messages that immediately follow a given one in the graph. Such information is filled in as the context graph is built and is used to determine when a message becomes stable, that is, when all its successors are known.

pred_msg_num: When a message is received from the network, it carries the *msg_num* of the predecessor message from each host. It is the job of one of the micro-protocols to fill in the *predecessors* attribute based on this information.

predecessors_needed: Used to keep track of how many predecessor messages have already been received by the framework. This is used to determine when a message can be added to the context graph.

successors_received: Similar to the above, this attribute is used to keep track of how many successors to a message have been received in order to determine when a message becomes stable.

stable: Whether the message is stable or not.

sorted: Marks messages whose position in the ordering has already been defined.

In addition, the framework provides the abstraction of a message bag, which holds all messages currently within the composite protocol, and can be accessed by all micro-protocols. Attributes are associated with messages right before they are entered into the bag, although some attributes may have their values set only later by a micro-protocol.

4.2 Events

The main events used by the atomic ordered multicast are listed below; for simplicity, we assume all events are blocking and sequential. Other events related to garbage collection procedures are not represented here.

MSG_FROM_APPLICATION(msg): Triggered as soon as the TSSM requests that a message be multicast.

BROADCAST_RECEIVED(msg): Triggered when a message is received from a lower level protocol due to a multicast from another host.

BROADCAST_EXECUTED(msg): Triggered each time a message is passed to the lower level protocol for delivery in multicast mode.

MSG_INSERTED_INTO_BAG(msg): Triggered by the framework each time a message is added to the bag.

PREDECESSORS_NEEDED(msg): Triggered if a micro-protocol verifies that any of a newly arrived message's predecessors have not been received yet.

PREDECESSORS_RECEIVED(msg): Triggered by the micro-protocol in charge of retransmission requests when the requested predecessors have been received.

MSG_ADDED_TO_GRAPH(msg): Triggered for each message that is added to the graph.

MSG_STABLE(msg): Triggered when an unsorted message becomes stable, enabling the sorting process to proceed.

MEMBERSHIP_STABLE(msg): Membership messages may have a different criteria for being declared stable, which makes a separate event necessary.

SUSPECT_HOST_DOWN(host): Triggered by the monitor micro-protocol when no message from a host is received during a monitoring interval.

MEMBER_FAILED(host): Triggered by the membership micro-protocol to inform others when a member is removed from the group.

There are also some events triggered by timer expirations, which are used when a micro-protocol has to wait for some time before taking an action. Those are usually private to a specific micro-protocol and will be mentioned when the micro-protocols using them are described.

4.3 Micro-protocols

We proceed now to describe the micro-protocols implemented for the atomic multicast composite protocol. When necessary, the protocols are presented as pseudo-code, whose features should be clear from the text. Some procedures may be mentioned without their code being presented, but their semantics should be clear from their names and contexts.

4.3.1 Validity

Each message delivered to the framework must be checked to make sure it is valid. A valid message must have a reasonable format and come from a host that is known to be in the group of currently active hosts. The micro-protocol must also detect and discard duplicate messages.

The action of this protocol is triggered by the events MSG_FROM_APPLICATION and BROADCAST_RECEIVED, which provide notification about messages entering the framework. Once a message is accepted as valid, its attributes are computed and it is inserted in the bag of messages provided by the framework. This in turn triggers the MSG_ADDED_TO_BAG event in other micro-protocols.

```
micro-protocol liveness
    var msg_sent: boolean;

    event handler HEARTBEAT_TIME
    begin
        if not msg_sent then
            msg = build_message( type = HEARTBEAT );
            insert_into_bag(msg);
        end
        msg_sent = false;
    end
    event handler BROADCAST_EXECUTED( msg )
    begin
        msg_sent = true;
    end

    initialize: set_timer_event( HEARTBEAT_TIME, repeat, "interval length" );
end liveness
```

Figure 5: Liveness micro-protocol

4.3.2 Liveness

Each message actually passed to the lower level protocol for delivery is seen by this micro-protocol when the BROADCAST_EXECUTED event gets triggered. Liveness keeps track of the last time a message was sent. If more than a given time passes without any message being sent, this micro-protocol is responsible for producing a *heart-beat* message in order to let other hosts know that it is still alive. The pseudo-code can be seen in figure 5.

The detection of a timeout is done by means of a LIVENESS_TIMEOUT event that is set and used by this protocol. The BROADCAST_EXECUTED event handler sets a flag each time a multicast is executed, and a LIVENESS_TIMEOUT event handler is triggered

periodically to verify the flag. If no multicast is executed during a whole interval, it builds a *heart-beat* message and adds it to the bag, which will trigger MSG_INSERTED_INTO_BAG and make the message available to other protocols that may have to handle it before it is actually multicast. That is done to ensure that each message sent carries information about the current state of the context graph.

4.3.3 Monitor

This is, in a certain sense, the "complement" of the previous protocol. It must keep track of all messages received from the lower-level protocol in order to verify which hosts have sent messages during the previous monitoring interval. Each time a message arrives, a MSG_INSERTED_INTO_BAG event handler marks the sender host as alive for the current interval.

A timer is set to trigger MONITOR_INTERVAL events periodically. The event handler then verifies if there were any hosts from which no messages were received during the last interval, and triggers a SUSPECT_HOST_DOWN event for each of them. This starts a membership agreement round to reach agreement on whether the hosts have really failed. This last task is performed by the membership micro-protocol. The pseudo-code for the monitor is shown in figure 6.

```
micro-protocol monitor
   var msg_received_from:   array of hosts;

   event handler MONITOR_INTERVAL( msg )
   begin
      for host = "all hosts in the group" do
         if msg_received_from[msg.attr.sender_id]
         then trigger( SUSPECT_HOST_DOWN, msg.attr.sender_id ); end
         msg_received_from[msg.attr.sender_id] = false;
      end
   end
   event handler MSG_INSERTED_INTO_BAG(msg)
   begin
      msg_received_from[msg.attr.sender_id] = true;
   end

   initialize:  set_timer_event( MONITOR_INTERVAL, repeat, "interval length" );
end monitor
```

Figure 6: Monitor micro-protocol

4.3.4 Membership

In this application, the membership protocol must be able to identify and remove failed hosts from the group, but not re-integrate them. Given these requirements, the protocol starts with an initial list of members provided at initialization time and just removes failed hosts from it as appropriate. As far as detection and removal of failed hosts is concerned, it implements the same membership protocol as used in Consul. This is, when a host receives a *suspect down* multicast message stating that a given host is suspected to be

down, it checks its monitor to determine if it has received any message from the suspect host during the last monitoring interval. If so, it immediately multicasts a reply message agreeing that the host is down (*ack* message); if not, it multicasts a *nack* message.

```
micro-protocol membership
   var group_members, suspected_down:  list_of_parts;

   event handler SUSPECT_HOST_DOWN(who)
   begin
      add_part( who, suspected_down );
      msg = build_message( type = SUSPECT_DOWN, suspect = who );
      insert_into_bag(msg);
   end

   event handler MSG_INSERTED_INTO_BAG(msg)
   begin
      if ( msg.attr.type == SUSPECT_DOWN ) then
         if ( heard_from_host(msg.attr.suspect) )
         then reply_type = NACK_DOWN;
         else reply_type = ACK_DOWN; end
         new_msg = build_message( type = reply_type, suspect = msg.attr.suspect );
         insert_into_bag( new_msg );
      end
   end

   event handler MEMBERSHIP_STABLE(msg)
   begin
      delete_part( msg.attr.suspect, suspected_down );
      if ( "all successors are ACK_DOWN" ) then
         delete_part( msg.attr.suspect, group_members );
         trigger( MEMBER_FAILED, msg.attr.suspect );
      end
   end

   export method member_list():  list_of_parts;
   export method valid_host( host ):  boolean;
   export method group_member( host ):  boolean;

   initialize:  "read list of group members from file"
end membership
```

Figure 7: Membership micro-protocol

A membership voting round is started by a host if the SUSPECT_HOST_DOWN event is triggered. The corresponding event handler assembles a *suspect down* message to be multicast. When such message becomes stable for membership purposes, a MEMBERSHIP_STABLE event triggers a handler that verifies replies. If all are *ack* messages, the group has agreed on the failure of the host, which is then marked as down and removed from the group. Otherwise, some host has received a message from it in the recent past and it is considered alive.

The code itself is quite simple, since the decision about when agreement is reached is transferred to the stability micro-protocol. Membership itself only gets activated when a host becomes suspect and when agreement is reached. The pseudo-code is in figure 7.

4.3.5 Context graph

The local view of the context graph is implemented by this micro-protocol. Its sole function is to receive incoming messages and insert them in the graph if possible. If any of the message's predecessors is missing, it transfers the message to the **reliability** micro-protocol, which is responsible for retrieving missing messages. When a pending message finally has all its predecessors in place, it is added to the graph. The pseudo-code is shown in figure 8

```
micro-protocol context_graph;
   var graph_leaves, oldest_not_stable:   array of msg;

   private method add_to_graph( msg );
   begin
      graph_leaves[msg.attr.sender_id] = msg;
      for host = "all hosts in the group" do
         predecessor = msg.attr.predecessors[host];
         predecessor.attr.successors[msg.attr.sender_id] = msg;
      end
   end
   event handler MSG_INSERTED_INTO_BAG(msg)
   begin
      msg.attr.predecessors_needed = "size of group";
      if ( msg.attr.direction == UP )
      then for host = "all hosts in group" do
            msg.attr.predecessors[host] = msg_get(msg.attr.pred_msg_num[host]);
            msg.attr.predecessors_needed--;
         end
      else for host = "all hosts in group" do
            msg.attr.predecessors[host] = graph_leaves[host]; msg.attr.predecessors_needed--;
         end
      end
      if ( msg.attr.predecessors_needed )
      then trigger( NEEDS_PREDECESSORS, msg );
      else add_to_graph( msg ); trigger( MSG_ADDED_TO_GRAPH, msg ); end
   end
   event handler PREDECESSORS_RECEIVED(msg)
   begin
      add_msg_to_graph( msg ); trigger( MSG_ADDED_TO_GRAPH, msg );
   end
   event handler MSG_STABLE(msg)
   begin
      host = msg.attr.sender_id; oldest_not_stable[sender] = msg.attr.successors[sender];
   end
end
```

Figure 8: Context graph micro-protocol

There are also some event handlers responsible for removing messages from the graph that have already been delivered to the application, but those have been omitted for brevity.

4.3.6 Reliability

Since the lower level transport protocol may not be reliable, messages may not be received by a host. Reliability is guaranteed by means of a micro-protocol responsible for issuing retransmission requests.

Each message has as a part of its header the list of its predecessors in the graph, that is, the number of the last message from each host that had been added to the graph of the sender before the message was multicast. When a new message is added to the bag, the context graph micro-protocol identifies the messages declared to precede the new one in the sender's graph. If any of the predecessors are missing, a request is issued to retrieve them.

```
micro-protocol reliability;
   var pending_msgs: list_of_messages;
   event handler MSG_ADDED_TO_GRAPH( msg )
   begin
      sender = msg.attr.sender_id;
      for pending = "all messages in pending_msgs" do
         if ( pending.attr.pred_msg_num[sender] == msg.attr.msg_num ) then
            pending.attr.predecessors[sender] = msg;
            pending.attr.predecessors_needed--;
            if ( not pending.attr.predecessors_needed ) then
               trigger( PREDECESSORS_RECEIVED, pending );
            end
         end
      end
   end

   event handler PREDECESSORS_NEEDED(msg)
   begin
      new_msg = build_message( type = RETRANSMIT, data = (msg_id, leaves) );
      send_out_of_band( new_msg, msg.attr.sender_id );
      schedule_timer_event( REQUEST_TO_SENDER, msg );
   end

   event handler REQUEST_TO_SENDER(msg)
   begin
      if ( msg.attr.predecessors_needed ) then
         new_msg = build_message( type = RETRANSMIT, data = (msg_id, leaves) );
         send_out_of_band( new_msg, group_multicast_id );
         schedule_timer_event( REQUEST_BROADCASTED, msg );
      end
   end

   event handler REQUEST_BROADCASTED(msg)
   begin
      if ( msg.attr.predecessors_needed ) then discard_msg( msg ); end
   end
end reliability
```

Figure 9: Reliability micro-protocol

Psync combined the context graph handling and retransmission requests in a single unit, but we decided to separate them because the policy used to retrieve missing messages

is not dictated by the context graph and can, in fact, assume various forms. The current implementation behaves as follows:

- When some predecessors of a message M are found to be missing, a request is sent directly to the host that originated M identifying M and the current leaves of the context graph in the local host.
- Upon receipt of a retransmission request, a host retransmits all messages in its graph between the messages mentioned as leaves in the request and M. It is guaranteed to have them all, otherwise they would not be in the context graph and therefore, would not have been considered predecessors to M.
- If no answer from the sender of M arrives in a defined interval, the host requesting the retransmission assumes it failed and multicasts the request to the group.
- If after some fixed time no answers to this multicast request are received, the message M is discarded, since there is no other host that can provide its predecessors.
- The reliability micro-protocol must pay attention to new messages added to the graph in order to identify those that may have been received due to retransmissions.

The pseudo-code for the policy just described is shown in figure 9.

It should be clear by now that this is just one of the possible policies for handling missing messages. One of the interesting features of the composite protocol approach is exactly that other policies may be tried easily simply by replacing the implementation of this micro-protocol.

4.3.7 Stability

We decided to make stability a separate micro-protocol due to the simplification achieved in the design of the other micro-protocols and because we can customize the criteria for a message to be considered stable. This micro-protocol can, with the same basic actions, identify as stable both general messages and membership agreement messages, although they have different constraints.

As discussed previously, a message is considered to be stable by a host when it receives messages from all other hosts that were sent in the context of that message (i.e., having that message as a predecessor). The implementation of the task is lengthy due to graph traversal operations, although simple to describe. There are two events that may make a message stable:

- **MSG_ADDED_TO_GRAPH:** When a message is added to the graph, its predecessors have one more successor defined. All messages that complete their set of successors become stable.
- **MEMBER_FAILED:** If a message already has successors from all other hosts except the one that failed, the removal of that host makes that message stable.

The implementation of this simple idea is complicated by the fact that many messages from many hosts may become stable at the same time. For the ordering enforced by the composite protocol to hold across all hosts in the group, the context graph must be traversed in a fixed way by all instances of the protocol, so that all of them perceive messages becoming stable in the same order.

As an example, one of the degrees of freedom for the implementation of stability is: should we make all messages from a host stable in sequence before messages from others, or should we try to make one message stable from each host in turn? This decision may impact the behavior of FT-Linda applications in different ways, so we intend to explore different approaches.

4.3.8 Ordering

The context graph provides only a partial order as defined by the causality relation among messages. Since messages in the graph may arrive at each node at different times, each host may see messages becoming stable in different orders. We need another micro-protocol that will use the stability information in each node to derive a total order common to all hosts. This is necessary since the partial order does not define relations between messages sent at the same logical time, that is, messages which do not have a dependency relation in the graph. In essence, then, we need a micro-protocol that imposes a topological ordering on the graph. This protocol has to be able to determine when it is safe to sort a portion of the graph available so far.

These requirements are achieved in the ordering micro-protocol by identifying the sub-graph containing all *non-committed messages*—that is, those that have not been delivered to the application—that do not have a direct dependency relation to a message that has just become stable, and then applying a topological sort to this sub-graph. A detailed analysis of this algorithm shows that this guarantees that the same total order will be seen by all hosts in the group [16].

Again, we intend to take advantage of the flexibility of the composite protocol model to study how different topological sorting schemes will affect the performance of the FT-Linda runtime system.

4.3.9 Other micro-protocols

There are a few additional micro-protocols that provide secondary aspects of the service. In particular, these are used to isolate tasks like retransmission of messages, garbage collection of data structures, and filtering of control messages (e.g., *heart-beat* messages) from the sorted stream.

5 Conclusions and future work

This paper has described the development of a customized atomic ordered multicast protocol to be used as the communication substrate for a new implementation of FT-Linda on a network. One of the most difficult issues when developing complex protocols like this one in normal environments is how to implement the interactions among the various abstract components. The event-driven protocol composition framework provides an elegant way to achieve a modular implementation, where abstract elements can be easily mapped into real code. This work provides a very good example of its capabilities. Although the algorithms and policies implemented are derived from Consul, the resulting implementation as a composite protocol offers a much simpler structure, where interdependencies and interactions may be modeled in a much clearer way.

Future work will include testing different policies for certain micro-protocols, such as ordering, reliability, etc. in order to determine which ones adapt better to the FT-Linda system. Other studies will be coordinated with future research involving FT-Linda itself.

As an example of the latter, it is our goal to add re-integration of failed hosts to FT-Linda, which will allow us to test even further the flexibility of the composite protocol approach. Host re-integration will create stronger requirements for the substrate, since it will have to be able to provide the reintegrated host with enough information to rebuild a consistent internal state. General solutions usually rely on checkpoint and message logging, both of which assume the existence of stable storage, so that failed hosts can restore their state to a point prior to the failure and then replay all subsequent messages in the system. It is our belief that a solution customized to FT-Linda can be implemented without the use of stable storage by providing the reintegrated host with the most recent view of the context graph and a copy of the tuple spaces at some fixed point in time. Implementing these new features will certainly benefit from the modular structure of the composite protocol framework.

References

[1] David E. Bakken. Supporting fault-tolerant parallel programming in linda. Technical Report TR 94-23, Computer Science Department, University of Arizona, Tucson, AZ, August 1994. Ph.D. Dissertation.

[2] David E. Bakken and Richard D. Schilichting. Supporting fault-tolerant parallel programming in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3), March 1995.

[3] Nina T. Bhatti and Richard D. Schlichting. Operating system support for configurable high-level protocols. Technical report, Department of Computer Science, University of Arizona, Tucson, AZ, USA, 1994. In preparation.

[4] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[5] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, 1990.

[6] Paolo Ciancarini. Distributed programming with logic tuple spaces. Technical Report UBLCS-93-7, Laboratory for Computer Science, University of Bologna, April 1993.

[7] Flaviu Cristian. Understanding fault-tolerant systems. *Communications of th*, 34(2):56–78, e ACM 1991.

[8] Dorgival O. Guedes and Osvaldo S. F. Carvalho. Um núcleo Linda para o desenvolvimento de aplicações distribuídas em uma rede unix. In *Proceedings of X Simpósio Brasileiro de Redes de Computadores*, Recife, PE, Brazil, April 1992. SBC.

[9] Matti A. Hiltunen and Richard D. 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, USA, October 1993.

[10] Matti A. Hiltunen and Richard D. Schlichting. Constructing a configurable group rpc service. Submitted to 15th Conference on Distributed Computing Systems, 1995.

[11] Matti A. Hiltunen and Richard D. Schlichting. Properties of membership services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, Phoenix, AZ, USA, April 1995.

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

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1:87–103, 1993.

[15] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Experience with modularity in consul. *Software — Practice and Experience*, 23(10):1059–1075, October 1993.

[16] Larry L. Peterson, N. C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7:217–246, 1989.

[17] David Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[18] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[19] Fred Schneider. Abstractions for fault-tolerance in distributed systems. In *Proceedings of the Tenth IFIP World Computer Congress*, pages 727–733, Dubli, Ireland, September 1986.

[20] Fred Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.