Semantics of Protocol Modules Composition and Interaction

Paweł T. Wojciechowski, Sergio Mena, and André Schiper

EPFL, School of Computer and Communication Sciences
1015 Lausanne, Switzerland
{First.Last}@epfl.ch

February 22, 2002

Abstract

This paper studies the semantics of protocol modules composition and interaction in configurable communication systems. We present a semantic model describing the x-kernel, Cactus and Appia — three frameworks that are used for implementing modular systems. The model covers protocol graph, session and channel creation, and inter-module communication of events and messages. To build the model, we defined a source-code-validated specification of a large fragment of the programming interface provided by the frameworks; we developed an operational semantics describing the behaviour of the operations through state transitions, making explicit interactions between modules. Developing the model and a small example implementing a configurable multicast helped us to better understand the design choices in these frameworks. The work reported in this paper is our first step towards reasoning about systems composed from collections of modules.

Contents

1 Introduction

Modularization is a well-known technique for simplifying complex communication systems. Here, we describe an approach which is based on implementing an application's individual properties as separate protocols, and then combining selected protocols using a software framework. This approach helps to clarify the dependencies among properties required by a given communication system, and makes it possible to construct systems that are customized to the specific needs of the application or underlying network environment. We are particularly interested in implementations of group communication infrastructure (or middleware), as configurability of protocols should be clearly required here; for example, different applications may demand very different properties and guarantees as far as the quality of service and failure semantics are concerned.

In this paper, we are primarily interested in the programming abstractions provided by the x-kernel [X96, HP91], Cactus [C01, WHS01] (successor of Coyote [BHSC98], which subsumes the x-kernel model), and Appia [A01, MPR01]. We have described an operational semantics of the programming interface offered by each framework, covering enough abstractions for expressing interactions between modules composed into a protocol graph. The frameworks also support primitives that can simplify the construction of protocols, such as support for processing messages, marshalling messages to the network format, and timeouts, but they are not covered here. We illustrate the model with a small program, implemented in Cactus and Appia.

We have chosen Cactus and Appia for two reasons. Firstly, each of the frameworks implements a very different approach to building configurable software, with a different range of programming abstractions. Therefore, it is interesting to look at each framework in turn. More importantly, we want a model that is general enough for building any kind of communication service, not just group communication. For example, Cactus has been used to implement many configurable protocols and services in distributed systems, such as Group RPC, real-time channels, secure communication service, and QoS components for CORBA. Appia has been used for the development of group communication and real-time protocols. On the other hand, systems such as Horus/Ensemble [E01, Hay98] have been designed to support modular and reconfigurable group communication, however the protocol stack can only be configured from selected protocols that use predefined event types.

The frameworks for building configurable services are highly concurrent with complex programming interfaces. This complexity makes it hard to achieve a clear understanding of the framework's behaviour based only on informal descriptions, in turn making it hard to build robust configurable systems. To the best of our knowledge, there exist only informal natural-language documents describing the x-kernel, Cactus, and Appia, covering the general architecture of each design and the programming interface but not precise enough or free from ambiguities; for example, we had to inspect source code on several occasions since the documentation was not clear enough.

Our work aims at precise understanding of the behaviour of programs that are implemented using these frameworks. An important question is about the sense in which the semantics of network subsystems composed from collections of protocol modules will relate to the behaviour of the actual implementation. This is an area that is often a secondary priority for the developers of practical module composition frameworks, yet is crucial to the long-term acceptance of this approach. While the work described here has not quite reached the point of reasoning about composition, it makes the important first step in this direction.

2 Architecture

The x-kernel, Cactus, and Appia frameworks have been designed for building flexible, configurable subsystems composed from collections of modules. These modules are called *protocols* but they are not necessarily restricted to networking software. The subsystem is constructed by selecting modules implementing the desired features, and composing them in a meaningful way. The composition of protocols is defined by a directed acyclic graph, in which nodes are protocols and directed edges define protocol dependencies. The dynamically created instances of modules are called *sessions*. The protocol interaction can be represented by a dynamic version of the dependency graph, in which nodes are the protocol sessions and edges define *communication channels*.

Figure 1 presents the architecture of the x-kernel, Cactus, and Appia. It can be noticed that the frameworks differ considerably in the way protocol modules (represented by boxes) are composed. The protocol sessions (ovals) communicate using messages or events, which are sent along communication paths (arrows). The protocols in Cactus are internally structured as collections of *microprotocols*. The protocol and microprotocol names are taken from our small example, which is described in Section 6.

The Cactus microprotocols communicate using events and shared data such as messages, with the events dispatching actions defined by event handlers; messages coming from outside the protocol session normally trigger an event. A protocol in the x-kernel and Cactus can create a new session dynamically, e.g. when a message arrives from a new participant. In Appia, all sessions must be created before a relevant communication channel is established (usually at the time when a protocol stack is configured).

The sessions (or protocols) in the x-kernel and Cactus decide themselves which other sessions are to receive a message — in the case of messages incoming from the network this information is usually extracted from the message.

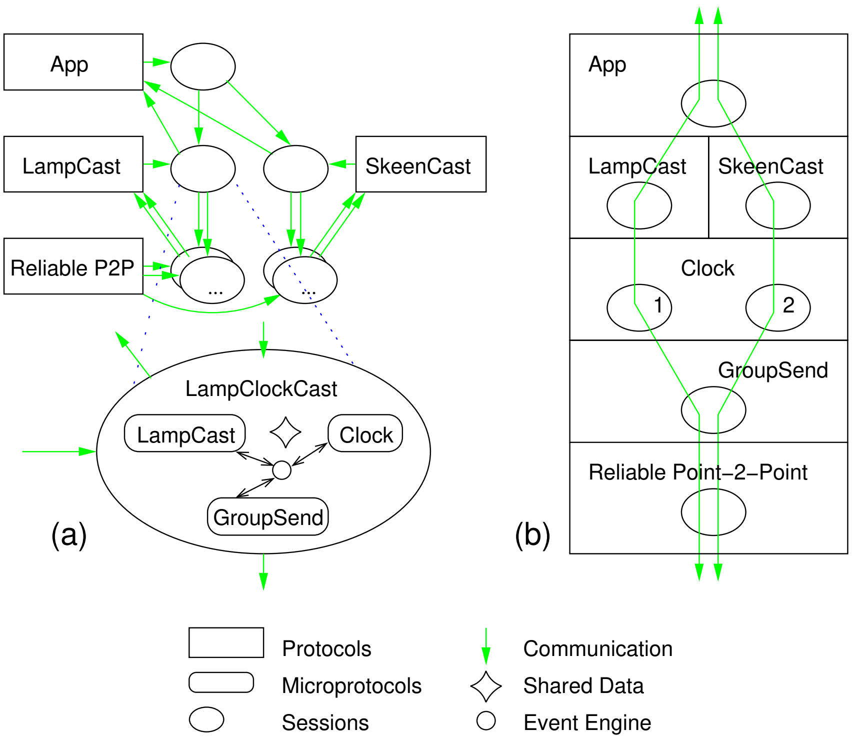

Figure 1: Example Protocol Composition in (a) X-kernel/Cactus, and (b) Appia

The message is forwarded to a next session by invoking an interface method. The message arrival to a session causes appropriate event handlers (within the session) to be invoked. In Appia, there is a scheduler which forwards events to sessions in the order which is defined by a communication channel; the channel name is extracted from the event.

In the following sections, we describe each framework in turn, giving semantics of the most important operations. We do not require from the reader knowledge of the formal semantics methods, but instead we use algebraic objects that should be also well known for non-theoreticians, such as sets, lists, tuples, maps, and relations. We introduce our notation when it is first used; see also Appendix B.

3 X-kernel

3.1 Protocols and Sessions

Protocols in the x-kernel form a hierarchy, informally called a "protocol stack", with one or more protocols at each level of the hierarchy. An instantiation of a protocol is called a *session*. It is quite reasonable to have several sessions of the same protocol active simultaneously, e.g. a different session for communicating with different remote protocol participant(s).

In order to implement some communication network service, sessions communicate messages with their *peer* participants, i.e. sessions executed by other kernels (likely to be on remote machines), which are at the same level of the protocol hierarchy. The communication between peers (except at the hardware level) is indirect, i.e. a message is passed to a lower-level session which in turn sends the message to its peer(s).

We use *Peer* to denote a set of participant names, ranged over by a, b. Each name unambiguously identifies one participant in the network. In practice, the participant names have some structure and contain, e.g. an IP address of the host which created the name, a protocol identifier, and any other data which is necessary to build a globally unique address. However, at the level of abstraction which we describe here these names can be pure names. Below, we also use the set *Protocol* of protocol names, ranged over by p, q, and the set *Session* of session names, ranged over by s.

Protocols are arranged into a directed acyclic *dependency graph*. Each protocol (i.e. a node in the graph) is aware of the protocols below it, on which it depends, but knows nothing about the protocols that may be arranged above it. The application code is at the top of the graph. Protocols which communicate outside are at the bottom.

If a protocol p sends messages to its peer(s) using a protocol q (i.e. p depends on q, denoted $p \searrow q$) then there is an edge from p to q in the graph, and we say that p is a higher-level protocol and q is a lower-level protocol. We define a dependency graph G as a finite set P of protocol names, together with a set E of ordered pairs (p, q), where $p, q \in P$ and $p \searrow q$, i.e.

$$G \quad := \quad (P, E) \quad \text{where } P \in \mathcal{S}(\mathit{Protocol}), E = \{(p, q) \mid p, q \in P \wedge p \searrow q\}$$

We use $\mathcal{S}(\mathit{Protocol})$ to denote all possible subsets of the set *Protocol* (i.e. $\mathcal{S}(\mathcal{T})$ is the powerset of $\mathcal{T}$, usually denoted $2^{\mathcal{T}}$). The protocol composition defined by graph G is *well formed* if G is free from dependency cycles, i.e. for all nodes p and q in G if $p \searrow\!* \, q$ then $\neg(q \searrow\!* \, p)$, where $\searrow\!*$ is a transitive closure of $\searrow$.

For creating new sessions, a protocol can either use an operation open, which creates a new session and returns its name to the caller, or an operation invite, which establishes a mechanism for accepting connections from participants. Subsequent connection requests from outside will create new sessions, and the original caller of invite is then notified of the new session name using $\mathsf{openDone}$. We define the following functions

$$\begin{aligned}
\mathsf{open} &: \mathit{Protocol}.(\mathit{Protocol} \cup \mathit{Session} \times \mathcal{L}(\mathit{Peer})) \rightarrow \mathit{Session} \\
\mathsf{invite} &: \mathit{Protocol}.(\mathit{Protocol} \times \mathcal{L}(\mathit{Peer})) \rightarrow () \\
\mathsf{openDone} &: \mathit{Protocol}.(\mathit{Session} \times \mathcal{L}(\mathit{Peer})) \rightarrow ()
\end{aligned}$$

where $\mathcal{L}(\mathit{Peer})$ denotes all possible lists of participant names such that each name is in *Peer*. By $\mathcal{T}$ 'dot', we specify the type $\mathcal{T}$ of the object on which a given function is invoked, e.g. open is invoked on behalf of a protocol. The function open accepts as arguments a protocol or session name together with a list of participants, and returns a session name.

To communicate messages, sessions use the following two operations.

$$\begin{aligned} \mathsf{push} &: \ \mathit{Session}.(\mathit{Message}) \to () \\ \mathsf{demux} &: \ \mathit{Protocol}.(\mathit{Message}) \to () \end{aligned}$$

Sessions invoke an operation push on behalf of some lower level session in order to send a message downward. Sessions can send a message upward by invoking an operation demux of a higher-level protocol which created the session. The higher-level protocol executing demux will pass the message to one of its sessions.

To express and maintain the internal state, we will need maps storing bindings from keywords of type $\mathcal{T}$ to values of type $\mathcal{T}'$. We represent maps using a set $\mathcal{M}(\mathcal{T} \mapsto \mathcal{T}')$ of all mappings from elements in $\mathcal{T}$ to elements in $\mathcal{T}'$, together with operations for adding and removing bindings from a map, and looking up an element.

To support routing of messages, each protocol p maintains two maps A_p and B_p, where $A_p \in \mathcal{M}(\mathcal{L}(\mathit{Peer}) \mapsto \mathit{Session})$ stores bindings of (at least two) communicating participants to their current local session of protocol p, and $B_p \in \mathcal{M}(\mathit{Peer} \mapsto \mathit{Protocol})$ stores bindings of a local participant to a higher-level protocol which invoked invite on behalf of p. The protocol p uses map A_p to look up a local session which is to handle a mesage communicated by a given set of participants. The B_p map is used when a message arrives from below and there is no session in A_p to handle the message. In this case, protocol p must first create a new session and pass to it the message with the names of session participants and the name of the higher-level protocol which called invite on behalf of p; the higher-level protocol's name is looked up in map B_p (it should be bound to the local participant's name).

3.2 Messages

Messages are objects communicated by sessions, which contain the user and protocol related data. For each message, a new thread is dispatched. The thread invokes protocol and session procedures on behalf of the message, so that the message "flows" in the protocol graph either upward or downward.

As the message is passed through sessions on the way down, message headers are added, the message may fragment into multiple messages, or the thread may suspend itself while waiting for a reply message. When the message reaches the lowest level session, it is marshalled and sent over network to remote participant(s). As the message is passed through sessions on the way up, headers are stripped, the thread may suspend itself while waiting to reassemble the message into a larger message, or the message may serialise itself with sibling messages.

The set *Message* is a set of messages, ranged over by m, n. A message can be represented as a list of headers. A header attached on a given level contains a list of participants that is used to identify a session to handle the message at the remote site, i.e.

$$Message \quad = \quad \mathcal{L}(\mathcal{L}(Peer))$$

We neglect any protocol-dependent data in headers and the message payload. As convention, we assume that the first element of each header denotes a name of the participant which is to receive the message. This name will be used to find a name of the higher-level protocol which should receive the message travelling upwards.

3.3 X-kernel Operations

3.3.1 X-kernel State.

We define the x-kernel state as a triple (A, B, M), where A and B are maps which contain bindings from protocol names in $P \in \mathcal{S}(Protocol)$ to maps A_p and B_p (defined in §3.1); and M is a map from active messages to sessions which are currently visited by the messages, i.e.

$$(A, B, M) \quad \in \quad \mathcal{M}(P \mapsto A_p) \times \mathcal{M}(P \mapsto B_p) \times \; \mathcal{M}(Message \mapsto Session) \; .$$

The state determines a set of semantic objects which have been introduced in the preceding sections, such as protocols, sessions, and messages. The purpose of this section is to describe how the x-kernel operations transform this state. The transformation is itself dependent on the graph G (defined in §3.1), which defines relation between protocols that have been used to build the protocol stack.

In order to describe behaviour of the operations supported by the programming interface (defined as a set of functions), we use a transition relation of the form $S, p \vdash op(n) \triangleright S'$, which means that the execution of operation op initiated or invoked by p in some state (or context) S leads to state S'; op has parameters n. Alternatively, we could use notation $op(n) : (S, p) \rightarrow (S', p)$. The state is represented by relevant set(s) of elements. In our case, the context of every transition relation is always a single protocol stack, i.e. S always describes (part of) the state of a local runtime system only.

Due to limited space, we omit operations which are used, e.g. for closing sessions, timeouts, and implementing *synchronous* protocols, i.e. protocols in which the caller blocks until a reply can be returned.

3.3.2 Session Creation.

Below, a protocol p or session k of p (such that there is some binding to k, denoted $(\circ \mapsto k)$, in p's map A_p of active sessions) creates a session of a lower-level protocol q by invoking an operation `open`, passing as arguments its name (which will be used for callbacks) and a complete list l of session participants — typically the names of a local and remote participant(s). By convention, the first element of list l is the name of a local participant, e.g. $a_{local} = \{p, \mathit{IP\text{-}addr}, \mathit{port}\}$.

$$\frac{\begin{array}{l} p \searrow q \\ k = p \vee (\circ \mapsto k) \in A_p = A[p] \\ \hfill l := a_{local} :: l_{remote} \hfill \end{array}}{G, A[q], k \vdash s := q.\mathsf{open}(k, l) \triangleright G, A[q] \oplus (l \mapsto s)} \quad (1)$$

This first constructs the list l (:: is a concatenation symbol to append a new element to a list), and invokes open which returns s. Here, we write $q.\mathsf{open}(k, l)$ to denote the fact that the operation is invoked on behalf of q, not inside the caller k. The state transition records a new local binding of l to s in q's map A of active sessions. As a follow up, protocol q (or its sessions) may also open one or more sessions of its own lower-level protocols. We use the following notation: $A[q]$ looks up q in map A and returns an element bound to q, in our case it is another map, say A_q; $A_q \oplus (l \mapsto s)$ returns map A_q with a new binding of l to s; if l was already bound in A_q, its previous binding disappears.

Alternatively, invite can be invoked to ask a lower-level protocol to create a new session at some future time (usually a time when a relevant incoming message arrives to the protocol and needs to be processed).

$$\frac{\begin{array}{l} p \searrow q \\ \hfill l := a_{local} :: \mathit{nil} \hfill \end{array}}{G, B[q], p \vdash q.\mathsf{invite}(p, l) \triangleright G, B[q] \oplus (l \mapsto p)} \quad (2)$$

Above, a protocol p invokes invite of some lower-level protocol q, passing its name and a list l of participants (which in this case typically contains only the local participant a_{local}). The binding of l to p is stored in a local map $B[q]$, so that when a new session of protocol q is created, q will be able to pass the name of the session to p (using $\mathsf{openDone}$).

3.3.3 Sending Downward.

A message is passed to a lower level session by calling the push operation on behalf of the session. Below a session s pushes a message m to session s'.

$$\frac{\begin{array}{l} M[m] = s \\ \hfill l_a := a_{local} :: l_{remote} \hfill \end{array}}{G, M, s \vdash s'.\mathsf{push}(l_a :: m) \triangleright G, (M \ominus m) \oplus (l_a :: m \mapsto s')}$$

The session s takes a complete list l_a of participants and invokes push on behalf of a session s', which has been created to deal with the messages of participants l_a at the lower level. The message m is passed in push with a new header l_a appended. To model migration, we record the new session of message m in map M, by first removing the old binding and then adding a new one ($M \ominus m$ returns map M without a binding of m).

Note that the session name s' was passed to the protocol of s when the session s' was created — either returned in open, or passed indirectly together with the list of participants l_a by the execution of callback $\mathsf{openDone}$ (as explained below).

3.3.4 Sending Upward.

When a message arrives to the x-kernel from the network, a kernel thread is dispatched which will *shepherd* the message in a direction from lower to higher level protocols and sessions (beginning from the lowest level).

A session s' can use an operation demux to send a message m upward to a session of a higher-level protocol p. (The name p was given to session s' when s' was created.) There are two behaviours of demux possible, depending on if there is a session of p which is to process message m, or it has to be first created. We describe these two cases in turn. They use an x-kernel operation $\mathsf{pop} : \mathit{Session}.(\mathit{Message}) \rightarrow ()$.

$$s' \vdash \boxed{p.\mathsf{demux}(m)\colon}$$
$$m = l_a :: n$$
$$\frac{A[p][l_a] = s}{G, M, p \vdash s.\mathsf{pop}(n) \triangleright G, (M \ominus m) \oplus (n \mapsto s)} \quad (1)$$

The rule (1) first extracts from message m a header l_a, which is a list of participants at the current level (:: is used here in the pattern expression to bind the first element of the list to a local name l_a), then it looks up the current session of l_a in a local map A of protocol p (here a session s is found), and finally an operation pop is used to pass to session s a fragment n of the original message, i.e. message m with header l_a stripped. From now on, the message is in session s, which we denote by modifying map of messages M.

Alternatively (2), if no session found in A then protocol p must create a new session which can process the message m, and notify a relevant higher-level protocol about the name of the session. The name of the higher-level protocol q can be found in map B, using as a keyword the first element of the message header l_a, i.e. the address of a local participant which is to receive the message.

$$s' \vdash \boxed{p.\mathsf{demux}(m)\colon}$$
$$m = l_a :: n, \quad \text{where } l_a = a_{local} :: l_{remote}$$
$$A[p][l_a] = \mathit{null}, \;\; B[p][a_{local}] = q$$
$$s \in \mathit{Session}$$
$$\frac{p \vdash q.\mathsf{openDone}(s, l_a)}{G, A[p], M, p \vdash s.\mathsf{pop}(n) \triangleright G, A[p] \oplus (l_a \mapsto s), (M \ominus m) \oplus (n \mapsto s)} \quad (2)$$

This rule is similar to rule (1) above but protocol p must first create a new session (let us call it s) and use openDone to pass to a higher-level protocol q (found in map B) the new session's name s and a complete list of session's participants. Then p passes message n to s as in rule (1). The state transition modifies M to record the fact that the message has migrated.

4 Cactus

Cactus extends the x-kernel hierarchical composition of protocols with fine-grain parallel composition. The Cactus protocols can be composed of semi-independent microprotocols, each of which implements a well-defined property or function of the protocol, e.g. appending and verifying a checksum in a message. Below we focus on Cactus/J, which is one of the prototype implementations of Cactus.

4.1 Microprotocols and Events

A *microprotocol* is a section of code, structured as a collection of *event handlers*, where an event handler is simply a procedure invoked with every occurrence of the event. We define *Microprotocol* as the set of microprotocol names, ranged over by x, y. The set *Handler* is the set of event handler names, ranged over by h. An *event* defines an occurrence that causes one or more microprotocols to be executed. For example, an event such as message arrival might trigger the event handlers of a microprotocol which detects host failures, and a microprotocol which is responsible for message ordering, etc. The events not only drive the flow of control, by executing event handler procedures associated with a given event, but also pass data from the trigger point to the handler. The set *Event* is the set of valid event names (or types), ranged over by e, e'. We denote the *occurrence* of an event e as a triple (x, e, v), where x is the caller which raised event e, and v is a value passed with the event.

In order to associate a handler with a particular event, a microprotocol invokes an operation

$$\mathsf{bind} \quad : \quad \mathit{Event} \times \mathit{Handler} \times \mathit{Int} \rightarrow ()$$

specifying the event name, the handler name, an integer which is used to determine an *order* in which handlers will be executed, and a static argument (omitted here) which is passed to the handler when an event occurs (this can be used to parameterize a handler and allow its use with more than one event types). Below a microprotocol x binds an event handler h to event e.

$$\frac{hl = E[e]}{E, x \vdash \mathsf{bind}(e, h, i) \triangleright E \oplus (e \mapsto \mathit{sort}_{\leq}((h, i) :: hl))}$$

This registers a handler h of event e in a map $E \in \mathcal{M}(\mathit{Event} \mapsto \mathcal{L}(\mathit{Handler} \times \mathit{Int}))$, which is part of the Cactus/J state of a composite protocol that contains x. The map E stores bindings from an event name to a list of event handlers which are to handle the event, where each handler name is paired with the order argument i; the list is ordered with increased i. The value i does not need to be unique for each handler; handler names with the same order argument are placed in an indeterminate order. With every occurrence of e, the handlers will be executed in sequence as they appear in the list $E[e]$. The function $\mathit{sort}_{\leq}(l)$ returns a list l sorted by partial order relation $\leq$.

4.2 Event Raising and Handling

An event can be raised by calling

$$\mathsf{raise} \quad : \quad \mathit{Event} \times \{\textsc{Sync}, \textsc{Async}\} \times \mathcal{T} \rightarrow ()$$

specifying the name of the event, the calling mode, and a dynamic argument, such as a message that is associated with the event. When an event is raised, handlers bound to this event execute sequentially in the specified order. Each handler is passed both the static argument defined at binding time and the dynamic argument. The calling mode μ is either Sync, which invokes handlers immediately and blocks the caller until the last handler is executed, or Async, which allows the caller to proceed concurrently with the handlers (the handlers can be executed after a specified delay, omitted here).

Below we define the behaviour of raise, assuming that a microprotocol x raises an event e with a dynamic argument v

$$\boxed{\mathsf{raise}(e,\mu,v)\text{:}}$$

$$\frac{\mu = \textsc{Sync} \vee \mu = \textsc{Async} \qquad E[e] = (h_1, \_) :: ... :: (h_n, \_) :: \mathit{nil}}{E, x \vdash \mathit{invoke}(h_1, v), ..., \mathit{invoke}(h_n, v) \triangleright E \wedge (x, e, v)_\mu \ \mathit{raised}}$$

This looks up in map E a list of handlers of event e and executes the handlers, passing v to each handler. The event raising is modelled by relation *raised*.

The caller x is either blocked until the last handler returns, or not, depending on mode μ. While the handlers of a particular event occurrence execute sequentially, it is important to note that they can execute concurrently with other occurrences of the same event or with other microprotocol code. Therefore access to any shared data should be synchronised.

4.3 Messages

Protocols in Cactus communicate using *messages*; a message is created by the application or a protocol session, and can travel through several layers of protocol sessions and across a network. Messages contain data stored in *attributes*, which can be accessed and modified by microprotocols. Message creation raises a predefined event *NewMessage*.

Below we use the set *Session* of session names, ranged over by s, and the set *Message* of message names (or references), ranged over by m, n. A message m is modelled as a triple of message attributes a, a message type $T \in \{\uparrow, \downarrow, \diamond\}$, and send votes V^m (the last two parameters are local to a session and never transmitted), i.e.

$$m \quad = \quad (a, T, V^m)$$

where each named attribute in a record a has a defined scope; it can be visible only in the current session, in the current stack, or within peer sessions only;

otherwise it is discarded or concealed. The message type T is equal $\uparrow$ if the message arrived from a session below, or $\downarrow$ if from above. The message type $\diamond$ is for a temporary message local to a session. The send votes V^m are described in §4.4.

A protocol can send a message to a Cactus session below or above using

$$\begin{array}{rcl} \mathsf{sendDown} & : & [\mathit{Session}\times]\mathit{Message} \rightarrow () \\ \mathsf{sendUp} & : & [\mathit{Session}\times]\mathit{Message} \rightarrow () \end{array}$$

where the first (optional) parameter is the name of a session to which the message is to be sent. If a message is sent to a non-Cactus session, e.g. to an x-kernel session, message attributes are converted into message headers by using a user-defined procedure (see also the push and demux operations in §3.3). Below we define the default behaviour of sendDown, assuming that a protocol session s sends a message m downward to a session s', created by Cactus/J.

$$s \vdash \boxed{\mathsf{sendDown}(s',m)\!:}$$

$$\textsc{down} \qquad \frac{s \vdash s'.\mathsf{fromAbove}(m)}{M, s' \vdash \mathsf{raise}(\mathit{MsgFromUser}, \textsc{Async}, m) \triangleright M \oplus (m \mapsto s')}$$

The session s invokes an operation fromAbove of the lower-level session s', passing m as the parameter. The execution of fromAbove raises asynchronously an event *MsgFromUser* which carries the message m. The message migration is recorded in map $M \in \mathcal{M}(\mathit{Message} \mapsto \mathit{Session})$ of active messages bound to their current sessions. Microprotocols which are interested in receiving messages from sessions above could handle the *MsgFromUser* event. Note, however, that any subsequent invocation of sendDown will also raise this event. Therefore, if the protocol requires to receive messages in a first-in-first-out order, some synchronisation is necessary so that the microprotocol handlers will be invoked in a sequence (e.g., in our example program, we have overwritten the operation fromAbove so that it executes a synchronous operation $\mathsf{raise}(\mathit{MsgFromUser}, \textsc{Sync}, m)$).

The semantics of message flow in the opposite direction is similar. The main difference is that sendUp calls either fromBelow of a specified higher-level session (inside which an event *MsgFromNet* is raised), or an operation demux (defined in §3.3.4) of a higher level protocol, if no session has been specified.

4.4 Message Events

An event can be associated with a particular message type ($\downarrow$,$\uparrow$). This event is triggered by a collective action of all microprotocols that have registered an interest, providing a way for microprotocols to agree upon event raising. To declare the interest, a microprotocol invokes

$$\mathsf{register} \quad : \quad \{\downarrow, \uparrow\} \times \mathit{Event} \rightarrow ()$$

passing a message type and an event name. Every subsequent creation of a message of that type has the potential of triggering the event. (If the event is to be caught, a bind call is also necessary.)

$$\frac{}{R_e^T, x \vdash \mathsf{register}(T,e) \triangleright R_e^T \oplus (x \mapsto \mathit{false}) \wedge (x,e,T)\ \mathit{registered}}$$

The invocation of register adds a new entry in a map R_e^T for message type T and event e. We mark registration by relation *registered*. The map $R_e^T \in \mathcal{M}(\mathit{Microprotocol} \mapsto \mathit{Boolean})$ is created dynamically and updated each time when some microprotocol executes operation register; it is a map from names of microprotocols to Boolean values (initially *false*) that represent the microprotocol "votes" signalling readiness of the event e to be raised for message type T (where T not equal $\diamond$).

For each message m, whenever message type T is assigned, Cactus/J uses maps R_e^T to build a (local to m) map V^m. For each event e that has been associated with the message type T, map V^m stores a copy of corresponding map R_e^T, i.e. $V^m[e] = \mathit{copyOf}(R_e^T)$. Each event e can be raised only once per message; that occurrence of e will pass name m to event handlers bound to e.

For each message, the message event is raised as soon as all of the interested microprotocols have called

$$\mathsf{signal} \quad : \quad \mathit{Message} \times \mathit{Event} \rightarrow () \ .$$

The signal operation requires to pass as arguments the names m of the message and e of the event which will carry the message. The behaviour of signal invoked by microprotocol x is below; the execution of signal should be atomic.

$$\frac{m = (a,T,V^m) \qquad V^m[e][x] = \mathit{false} \wedge \forall y \neq x\ V^m[e][y] = \mathit{true}}{E, V^m[e], x \vdash \mathsf{signal}(m,e) \triangleright E, V^m[e] \oplus (x \mapsto \mathit{true}) \wedge (x,e,m)_{\text{ASYNC}}\ \mathit{raised}} \quad (1)$$

$$\frac{m = (a,T,V^m) \qquad \exists y \neq x \mid V^m[e][y] = \mathit{false}}{E, V^m[e], x \vdash \mathsf{signal}(m,e) \triangleright E, V^m[e] \oplus (x \mapsto \mathit{true}) \wedge (x,e,m)\ \mathit{signalled}} \quad (2)$$

Rule (1) checks if x signals e for the first time and if all other microprotocols set their "vote" to raise event e associated with the message. If so, the event is raised asynchronously and all event handlers which have been bound to this event will receive the name of the message (see §4.2 for details). Otherwise (2), event e cannot be raised and we only set in V^m the message readiness as far as microprotocol x is concerned (and mark that the relation *signalled* holds).

For example, we can use this mechanism to implement a collective sending by several microprotocols. Below, we have two microprotocols x and y which share a message m and want to agree when to invoke an event carrying this message.

$$\frac{\begin{array}{l}(x,e,\downarrow)\ \textit{registered}\\ (y,e,\downarrow)\ \textit{registered}\\ m=(a,\downarrow,V^m)\\ \qquad\qquad (y,e,m)\ \textit{signalled}\end{array}}{E,x\vdash \mathsf{signal}(m,e)\triangleright E\wedge(x,e,m)_{\textsc{Async}}\ \textit{raised}}$$

We assume that microprotocols x and y registered their interest in raising an event e when a message of type $\downarrow$ will be received from above by the composite protocol. We also assume that some message m of this type eventually appeared and was handled and signalled by microprotocols x and y. Since x is the last microprotocol which signalled the readiness of message m, therefore it causes event e to be raised. A microprotocol (more precisely one of its event handlers) which has been bound to event e can now be invoked and, e.g., it might send the message out of the composite protocol.

5 Appia

A *protocol* in Appia consists of two static parts, one is called *layer* and the other one is called *session* (not to be confused with a session in the x-kernel and Cactus). Protocols interact using one or more coordinated channels. A *channel* defines routing of events across protocols, and is defined by a set of instances of sessions (i.e. objects of class "Session").

5.1 Layers and Sessions

A *layer* declares types of events which are either generated, required, or accepted by the protocol. Appia uses the event declarations to verify partial correctness of QoS definitions (we describe this verification below). A layer is also used to create instances of its session. A *session* implements the actual protocol code, in particular it generates and handles events which have been declared by the corresponding layer. An event may carry a message. Messages can be marshalled and communicated in a network.

The set *Layer* is a set of layer names, ranged over by l. The set *Session* is a set of session names, ranged over by s. The name of a layer identifies unambiguously a protocol whose definition the layer is part of (so we may sometimes use terms "layer" and "protocol" interchangeably). A layer can use an operation

$$\mathsf{createSession} \quad : \quad \textit{Layer} \to \textit{Session}$$

to create many instances of its session (the name of the layer is passed as the operation argument).

Below we use a set $P \in \mathcal{S}(\textit{Layer})$ of names of all protocols/layers which are used to form a given protocol stack. In the context of P, we define the following three maps E_{g}, E_{r}, and E_{a}, which store bindings from layer names to, respectively, a set of types of events which are generated by a protocol, types of events which are required by the protocol, and types of events which are accepted by the protocol (that includes the former set), i.e.

$$E_{\mathrm{g}}, E_{\mathrm{r}}, E_{\mathrm{a}} \in \mathcal{M}(P \mapsto \mathcal{S}(\mathit{EventType})), \quad \forall l \in P \ E_{\mathrm{r}}[l] \subseteq E_{\mathrm{a}}[l]$$

where *EventType* is a set of abstract event types. A protocol l declares some event type T to be in $E_{\mathrm{a}}[l]$ but not in $E_{\mathrm{r}}[l]$ if the absence of events of this type is not critical for the protocol execution; therefore we could use protocol l to build protocol stacks which are meaningful even if events of type T are never generated in these stacks.

The Appia state contains set P of layers which are used to form a single protocol stack, together with a map $S \in \mathcal{M}(P \mapsto \mathcal{S}(\mathit{Session}))$ from layer names to sessions created by the layers. New sessions are created as follows.

$$\frac{l \in P}{P, S, l \vdash s := \mathsf{createSession}(l) \triangleright P, S \oplus (l \mapsto S[l] \cup \{s\})}$$

This transforms the state at a time when the protocol stack is initiated, recording a new session s created by layer l in map S.

In the following sections, we introduce other parts of the state such as channels and events, and show how the Appia operations transform the state at the normal protocol execution (i.e. we neglect any misbehaviour due to the omission of class definitions, or security attacks).

5.2 QoS Definitions and Channels

A *QoS definition* is simply a static list of layers, which is used to create a communication *channel.* The Appia framework partially verifies each QoS definition, checking if events that the layers declared as required are also declared as generated. The verified QoS definition is used to build a channel with blank slots; the slots can be filled as appropriate with sessions that are created by the layers.

A channel defines the flow of events through the sessions. Each channel maps layers from the QoS definition into concrete sessions which have been created by the layers. By selecting appropriate channels for routing different events through the protocol stack, an application can obtain a requested *quality of service* (QoS).

We model a QoS definition as a list of names of layers which are used to build a single protocol stack. A QoS definition $\mathit{qos} \in \mathcal{L}(P)$ constructed using protocols from P is *well formed* if for each event type T required by each protocol l from set L (of all elements from list qos) there exists some protocol l' in L which declared T in set $E_{\mathrm{g}}[l']$ of types of events generated by l'. We define relation *well-formed* for QoS definitions as follows.

$$\mathit{well\text{-}formed} \subseteq \mathcal{L}(\mathit{Layer})$$

$$\frac{S = \{\},\ L := \mathit{setOf}(qos) \qquad \forall l \in L\ \forall T \in E_{\mathrm{r}}[l]\ \exists l' \in L \mid T \in E_{\mathrm{g}}[l']}{P, S \vdash qos\ \mathit{well\text{-}formed}}$$

This verification is usually done before any session is created. The set $Q \in \mathcal{S}(\mathcal{L}(P))$ of QoS definitions, such that each definition is well formed can be used by Appia to create channels.

A set $\mathit{Channel} = \mathcal{L}(\mathit{Session} \times \mathit{Layer})$ is a set of channels, ranged over by c. Let $C \in \mathcal{M}(\mathit{Id} \mapsto \mathit{Channel})$ be a mapping from channel identifiers to channels in a given protocol stack, where a single channel c in map C is modelled as a list of session names paired with names of the corresponding layers in the protocol stack, i.e.

$$c \quad = \quad (s, l) :: t \quad \text{where} \quad l \in P,\ s \in S[l] \ .$$

The channel identifiers are unique per protocol stack; they are used by messages to identify a (corresponding) channel on a remote site that should be chosen to deliver the messages to peers.

Here is how a new channel is created and bound to sessions (first by user-defined binding and then automatic binding).

$$\frac{c := \mathsf{createUnboundChannel}(\mathit{ID}, qos) \wedge qos\ \mathit{well\text{-}formed}}{P, S, C \vdash c = \mathit{defaultBind}(\mathit{userBind}(c)) \triangleright P, S, C \oplus (\mathit{ID} \mapsto c)}$$

This first creates a new channel c from a well formed QoS definition qos using an Appia operation $\mathsf{createUnboundChannel} : \mathit{Id} \times \mathcal{L}(\mathit{Layer}) \to \mathit{Channel}$. The channel is identified by a fresh name $\mathit{ID} \in \mathit{Id}$. The new channel is initially *unbound*, i.e. each element (s, l) of c has a session name s equal null. After the channel is filled with sessions, a mapping of ID to the channel is recorded in map C.

In order to bind the free slots of an unbound channel to sessions that are created by corresponding layers (of the qos definition), the following two procedures are used. The first procedure must be set up by the protocol programmer, who can specify in this way which channels should share a common session.

$$\boxed{\mathit{userBind}(c)\text{:}}$$

$$\frac{c = (\mathit{null}, l) :: t \quad \text{where} \quad l \in P \qquad \exists s \in S[l] \mid s\ \mathit{required\text{-}by}\ c}{P, S, C \vdash \mathit{return}\ ((s, l) :: \mathit{userBind}(t)) \triangleright P, S, C}$$

This binds free slots in channel c to some existing sessions s, which are selected by a programmer from set $S[l]$. We assume that the sessions have been created before with $\mathsf{createSession}$. The sessions s are likely to be bound already to some other channels, so that they can process different types of events which originate from different channels. The choice of sessions is application-dependent; here modelled by relation *required-by*. If the relation does not hold, *null* slot is left.

The free slots that have not been bound explicitly by *userBind* are bound automatically by a default procedure below.

$\boxed{\mathit{defaultBind}(c)\text{:}}$

$c = (\mathit{null}, l) :: t \quad \text{where} \quad l \in P$

$$\frac{s := \mathsf{createSession}(l)}{P, S, C \vdash \mathit{return}\ ((s,l) :: \mathit{defaultBind}(t)) \triangleright P, S \oplus (l \mapsto S[l] \cup \{s\}), C}\ (1)$$

$$\frac{c = (s,l) :: t \wedge s \neq \mathit{null}}{P, S, C \vdash \mathit{return}\ ((s,l) :: \mathit{defaultBind}(t)) \triangleright P, S, C}\ (2)$$

This creates a new session s for each session-free layer l in a channel c and returns the channel with free slots filled with the session names.

A *protocol stack* is defined as a composition of all protocols that share (transitively) some communication channels. We define F to be a *well formed* set of channels where well-formedness means that each channel in F (built from a well-formed QoS definition) shares at least one session (selected by the user) with some other channel in the protocol stack. Formally, we can define the *well-formed* relation as below.

$$\mathit{well\text{-}formed} \subseteq \mathcal{S}(\mathit{Channel})$$

$$\frac{F = \{c\}}{P, S, C \vdash F\ \mathit{well\text{-}formed}}\ (1)$$

$$\frac{\forall c \in F\ \exists (s,l) \in \mathit{setOf}(c)\ \exists c' \in F \mid s\ \mathit{required\text{-}by}\ c' \wedge c' \neq c}{P, S, C \vdash F\ \mathit{well\text{-}formed}}\ (2)$$

where rule (1) denotes the fact that a set F with a single channel is also well formed. We represent a protocol stack as a map C from channel identifiers to channels which are taken from set F.

An application built on top of Appia as well as any external protocol used to communicate messages between different Appia runtime systems must be represented in the protocol stack by separate layers, providing a suitable communication interface. If we construct a graph from a protocol stack so that nodes are layers paired with session instances, and edges are created by mapping channels on nodes, then the top-level node(s) are used to inject messages from/to application(s) and the most bottom level would be used to interface Appia with the network.

5.3 Routing Table

After channels have been created, Appia can use information about the channels and events declared by protocols to construct an optimal routing path for each event type that is associated with a given channel.

We model a *routing table* as a map $R \in \mathcal{M}(Id \times EventType \mapsto \mathcal{L}(Session))$ from channel identifiers paired with types of events to routing paths, where a *routing path* is a list of sessions (ordered from top to bottom) which accept these events. A session *accepts* an event of type T if the session was created by a protocol which declared T in its set of accepted events (in map E_{a}, which has been defined in §5.1).

The map R is created from *all* routing paths which are well formed. A routing path $r \in \mathcal{L}(Session)$ of events of type T that are to travel in a channel identified by ID is *well formed* if r is a list of sessions constructed from a superset of sessions taken from channel c identified by ID, so that each session in r accepts events of type T and the order of sessions in r is the same as order of sessions in c. This can be defined formally as below

$$\mathit{well\text{-}formed} \subseteq \mathcal{L}(Session)$$

$$\frac{\begin{array}{l} C[ID] = c \\ \forall s \in setOf(r) \exists (s,l) \in setOf(c) \mid T \in E_{\mathrm{a}}[l] \\ \forall s, s' \in r \ \text{ if } s' = next(s,r) \text{ then } c = ... :: (s, \_) :: ... :: (s', \_) :: c' \end{array}}{P, S, C, R \oplus ((ID,T) \mapsto r) \vdash r \ \mathit{well\text{-}formed}}$$

where function $next(s,r)$ returns the first element following the first occurrence of s in list r. Routing paths are kept unchanged during system lifetime.

5.4 Events and Messages

Events are the only mean which can be used by protocol sessions (including the application session) to communicate with other sessions in the protocol stack. Messages are specialised events which can be marshalled and sent over network to remote sites; they contain headers with protocol-dependent data. The set *Event* is the set of valid event (and message) names, ranged over by e.

An *event* (or *message*) $e \in Event$ is represented as a tuple (T, ID, r, n), where T is the event type, ID is the name of the channel carrying events of type T, r is the list of sessions to be visited by e (which is built from the channel), and n is the event content. We say that a channel l *carries* (or *accepts*) events of type T if the QoS definition used to create the channel contains at least one layer l, such that $T \in E_{\mathrm{a}}[l]$. The event content n has two components *attrs* and m (denoted $n = attrs + m$), where *attrs* is the record (with named fields) of event attributes, and m is the list of message headers (attached to e by visited protocols). The m fragment is marshalled and sent over network together with T and ID. If e is not a message then m is empty; if e is a message then two attributes s and d of *attrs* are predefined and should contain the source and destination of the message.

Before a message of type T which arrived from a network can be injected into a local channel c identified by ID, it must be first verified (by a user-defined procedure) and then "wrapped" by one of the event tuples below.

$$\begin{array}{rcl} e^{\downarrow} & := & (T, ID, R[(ID,T)], n) \\ e^{\uparrow} & := & (T, ID, reverse(R[(ID,T)]), n) \ . \end{array}$$

The event tuples contain local routing data, which is found in R. The routing data will not change during e's lifetime. The choice between tuples $e^{\downarrow}$ and $e^{\uparrow}$ depends on if the event/message uses channel ID to travel downward, or upward ($reverse(l)$ returns a reversed list l). The verification procedure should check if T is accepted by channel ID.

5.5 Event Scheduling and Routing

Below we confuse events and messages for simplicity, and describe the flow of messages in a channel, modelled by modifications to a map of events $E \in \mathcal{M}(\mathit{Session} \times \mathit{Id} \mapsto \mathit{Event})$ from channel sessions to events.

A session s holding an event $e = (T, ID, r, n)$ can pass it along a channel identified by ID by invoking an operation $\mathsf{go}(e)$.

$$\textsc{down/up} \qquad \frac{E[(s, ID)] = e = (T, ID, r, n)}{C, E, \Phi, s \vdash \mathsf{go}(e) \triangleright C, E \ominus (s, ID), \Phi \cup \{(s, e)\}} \quad (1)$$

This transfers control to a (default or user-defined) scheduler ϕ, modelled as a set Φ of events paired with their last visited session, together with a function *takeEvent*, which returns one element from the set. We record the change of state by modifying the map of events and the scheduler set.

The scheduler ϕ selects an event e from Φ (the choice depends on the implemented scheduling algorithm), and passes e to the next session to be visited by the event.

$$\textsc{down/up} \qquad \frac{\begin{array}{l}(s, e) = \mathit{takeEvent}(\Phi) \\ e = (T, ID, r, n) \\ \qquad\qquad s' = \mathit{next}(s, r)\end{array}}{C, E, \Phi, \phi \vdash s'.\mathsf{handle}(e) \triangleright C, E \oplus ((s', ID) \mapsto e), \Phi \setminus (s, e)} \quad (2)$$

This selects an event e together with its last visited session s from Φ, and uses s to find out which is the next session s' to visit by e according to the routing path r (which has been extracted from the event tuple). It then invokes an operation handle of session s' to handle event e. We record the change of state by modifying a map of events, and removing (s, e) from the scheduler set. The handle operation will recognise a type of e and invoke a user-defined procedure to handle e. For simplicity, we assume in the rule above that a session can only hold one event at a time.

The scheduling of events depends on the event scheduler. The default policy is such that each two events which are initially processed by some session in a certain order (e.g. defined when the events are injected into a protocol stack by an application, or received from the network) will never be processed in the opposite order by any other session in the protocol stack. This implies that the whole protocol stack (i.e. all channels) behaves like a distributed queue which holds a first-in-first-out property.

6 Example Protocol Decomposition

To experiment with Appia Kernel v1.0 and v1.3 (Protocols v0.5) and Cactus/J 2.0, we have implemented in each of these frameworks a small example application that uses two communication services. The first service (AB) sends a message atomically to all processes in a distributed system and guarantees Atomic Broadcast [HT94]; it has been implemented by a mild modification to Lamport's mutual exclusion algorithm [Lam78]. The second service is an Atomic Multicast (AM), which sends only to a specified group of processes using Skeen's algorithm [Ske85]. We have decomposed each service into several modules, each implementing a small protocol, so that some modules in the protocol graph can be shared by the two services (in a given system). The modules are presented in Figure 1. The Atomic Broadcast algorithm and pseudocode of an example modular implementation in Appia and Cactus are described in Appendix A. We have set up enough semantics in this report to be able to understand the pseudocode.

In Cactus/J, we have decided to place modules *LampCast* and *Clock* in one composite protocol so that they can share a clock variable C_i, which both modules need to read (see Figure 1a); we did the same for modules *SkeenCast* and *Clock* (not shown in Figure). We might experiment with even finer grain protocols; e.g. the *LampCast* module could be further decomposed into two "microprotocols", one for receiving an application message, and the second one for receiving an acknowledgement message.

The clock variable in module *Clock* of the Appia implementation is not shared by other protocols. Therefore, we need to create a specialised event *ClockEvent* (c) in order to propagate the current clock value to *SkeenCast*, each time a new message arrives from the network. Also, we need to create another specialised event *TimeEvent* (t) carrying the message timestamp that is required by *LampCast*. If a given specialised event will be actually delivered depends on which channel is used to propagate the event. For example, if *ClockEvent* has been created by *Clock*'s session 1 which forms a channel together with a session of protocol *LampChan* (and sessions of *App* and *GroupSend*) then the system will silently discard this event since type *ClockEvent* is not accepted by any of the protocols that use this channel. The events are illustrated in Figure 2, where events marked with a dashed line are discarded.

Notice that each local event must be propagated upward *before* the event of type *Msg* carrying the message (types *AppMsg*, *AckMsg*, and *GroupMsg* are all subtypes of *Msg*). Unfortunately, we cannot pass the clock and timestamp values between modules using network messages since the message headers can only be accessed at the level on which they have been created by a peer participant (e.g. a header which contains the timestamp required by *LampCast* is stripped by layer *Clock*). Also, for sanity reasons, message attributes should not be used for this either since, e.g. the current clock value is required only by *SkeenCast* — it does not seem reasonable to extend the message format to include this value because we want to be able to remove or replace module *SkeenCast* at any time, however the format of network messages should not change so often

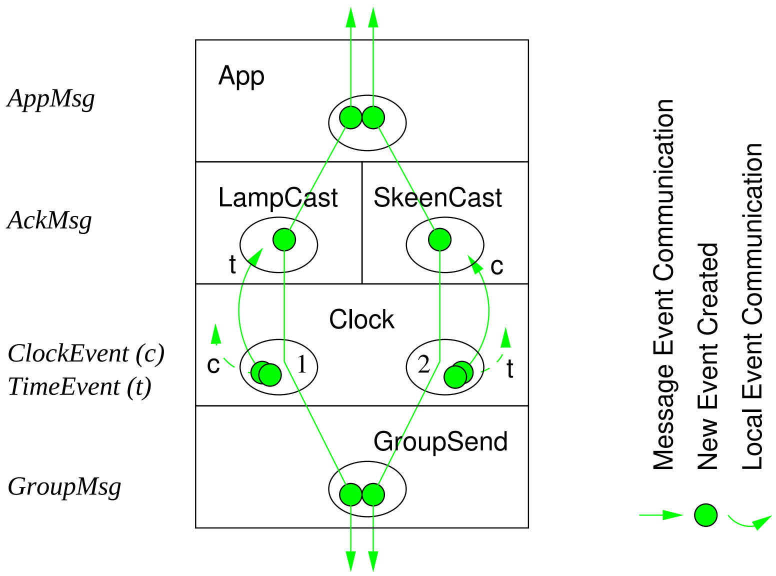

Figure 2: Example Decomposition in Appia: Atomic Broadcast

(the same for the timestamp value and *LampCast*).

7 Brief Comparison

Cactus supports fine-grain composition of microprotocols, which communicate using events or shared data. A composite protocol (built from a collection of microprotocols) can also be composed with other (composite) protocols, forming a protocol graph. This two-level architecture allows to decompose a given service in an arbitrary way. Appia offers less flexibility of the composition — modules are composed into a graph, and the pattern of communication between modules is restricted by the communication channels. The channels are static, optimised routing paths in the protocol stack.

In Cactus, the idea is that each well-defined property or function of a protocol could be implemented as a microprotocol. However, we need more experience to attain confidence when such fine-grain composition would be justified. In particular, increasing the number of concurrent microprotocols per composite protocol (which have to share resources) may increase the number of mutual dependences, in turn making it harder to notice possible deadlocks.

Appia supports partial evaluation of the protocol composition — for each communication channel it can verify if events declared as required are also generated. This helps to reject protocol compositions which are clearly not meaningful, however, of course it does not guarantee correctness. This simple evaluation could be improved if a programmer was able to specify some additional (application-dependent) constraints when defining a module, e.g. a requirement that *all* modules below in the communication channel should declare some event(s) as accepted, or required. For example, a fragmentation protocol must first compute the maximum size of message fragments which will be sent to the network. Since we may want to use the protocol with many different protocol stacks, the protocol has to first learn from all protocols below what is the size of the message headers which they use. Thus, *all* protocols below should handle an event (generated by the fragmentation protocol) which collects these data, otherwise the stack is not correct. In the context of Cactus, Hiltunen [Hil98] developed a methodology which is based on identifying relations between modules that dictate which combinations are correct; a configuration tool based on these relations allows only correct configurations to be created.

Some applications may require dynamic reconfiguration of the protocol stack. Cactus allows microprotocols to be loaded during execution and it is possible to rebind events to new (compatible) event handlers within a single composite protocol. Therefore, the configuration of modules can be changed on the fly (though we did not yet experiment with this feature). There is currently no similar support in Appia.

8 Related Work

To the best of our knowledge there is no other work that models the behaviour of the x-kernel, Cactus, or Appia operations. An understanding of the behaviour is critical for actually programming with these frameworks. In the Ensemble project, formalisation using the Nuprl theorem prover provided insight into the structure of the layered protocols and their optimization [Hay98], however the framework itself has not been described formally. There has been work on formalisation of modules composition, e.g. [SS01], however it further abstracts away from programming frameworks.

The approach of Serjantov *et al.* [SSW01] is similar to ours in that they aim to model the behaviour of partial systems, making explicit the interactions that the infrastructure offers to applications. They constructed an experimentally-validated specification of the standard UDP/ICMP sockets interface, including loss and failure, and integrated the above with semantics for an executable fragment of a programming language (OCaml) with OS library primitives. The UDP/ICMP network infrastructure and socket-based application correspond in our case to the *protocol framework* and *communicating protocols* implemented using the framework, with correspondingly more complex dependencies and mutual interactions. However, unlike them we do not need to deal with the distributed phenomena and complex failure semantics.

The goals put forth in [SWP99] in the area of the location-independent communication for mobile agents are also related to the approach described here in the sense that the choice or design of protocols must be somewhat application-specific. However, unlike the Nomadic Pict programming language [WS00, Woj00] which has been implemented and used to design many different communication infrastructures, provided as encodings of the high-level language primitives, the frameworks described in this paper use standard language facilities and support multi-level protocol composition.

9 Conclusion

9.1 Contribution

We have given a mathematically precise and experimentally validated model of protocol modules composition and interaction in Appia and Cactus (including part of the x-kernel programming interface). It has been illustrated with a simple example application that uses two (idealised) group communication algorithms. The model consists of a set of inference rules defining operations and state transitions. The contribution of the formalisation is twofold. It provides a clear and concise description of a fragment of the programming interface provided by each framework. Moreover, we think that this specification is at the right level of abstraction to help reasoning about the design differences — it describes the frameworks' behaviour (sufficiently accurately) but without going into too many implementation details. The specification is also precise enough to give some useful hints for the designers and implementors of such systems. However, the model is not complete — our primary goal was to understand the design features of the example frameworks, instead of developing concrete reasoning tools that could be applied for programs in Cactus or Appia. Nevertheless, it might be interesting to see how we could express and verify certain properties in this model, like for instance deadlock freedom. Due to lack of time, we also did not cover the whole programming interface and some operations are missing, e.g. for dealing with timeouts and dynamic microprotocol loading; also the description of threads, error situations, and event scheduling should be sufficiently covered. Developing and refining a small example application identified a bug in one of the frameworks, which has been fixed up in a newer release of the system.

9.2 Further Research

The work described in this paper is a step towards a better understanding of protocol modules composition and interaction. However, it provides only a starting point — much additional work is required on algorithms decomposition, semantics, and implementation. We hope to address some of this within our Crystall project, that aims at the design of group communication services with solid semantics foundations. In our future work, we would like to design a language with clean abstractions for module composition and interaction in the context of fault-tolerant computing. One way of making an application tolerant to partial failures, is to replicate its services on different machines using group communication algorithms. The goal is to decompose the algorithms into configurable modules in such a way that module dependencies are reduced, and the (internal) communication between modules is optimised. The language should adopt a model which allows an application to specify its requirements so that they can be adequately reflected by a protocol suite built from modules. The language should also support a type system that can be used to verify certain properties of the protocol suite. Eventually, it should be possible to integrate the language abstractions with standard frameworks that are used to build component based software, in order to increase the applicability of the method.

Acknowledgements We would like to thank Rick Schlichting and anonymous reviewers for useful comments that helped us to improve the paper. We also thank Luis Rodrigues for explaining some details of the Appia implementation. The project is supported by EPFL grant "Semantics-Guided Design and Implementation of Group Communication Middleware".

References

[A01] *The Appia project*, 2001. Documentation available electronically at `http://appia.di.fc.ul.pt/`.

[BHSC98] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: a system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, February 1987.

[C01] *The Cactus project*, 2001. Documentation available electronically at `http://www.cs.arizona.edu/cactus/`.

[E01] *The Ensemble project*, 2001. Documentation available electronically at `http://www.cs.cornell.edu/Info/Projects/Ensemble/`.

[Hay98] Mark Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.

[Hil98] Matti A. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, October 1998.

[HP91] 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.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[MPR01] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of ICDCS'01: The 21st International Conference on Distributed Computing Systems*, April 2001.

[Ske85] D. Skeen. Unpublished communication. Referenced in [BJ87], 1985.

[SS01] Purnendu Sinha and Neeraj Suri. On simplifying modular specification and verification of distributed protocols. In *HASE'01*, October 2001.

[SSW01] Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai)*, October 2001.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: A two-level architecture. In *Internet Programming Languages (ICCL '98 Workshop, Chicago, USA, May 1998)*, volume 1686 of *Lecture Notes in Computer Science*, pages 1–31. Springer, 1999. Also appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

[WHS01] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. A configurable and extensible transport protocol. In *Proceedings of the INFOCOM 2001*, April 2001.

[Woj00] Paweł T. Wojciechowski. Nomadic pict: Language and infrastructure design for mobile computation. Technical Report 492, Computer Laboratory, University of Cambridge, June 2000.

[WS00] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency. The Computer Society's Systems Magazine*, 8(2):42–52, April-June 2000.

[X96] *The X-kernel project*, 1996. Documentation available electronically at `http://www.cs.arizona.edu/xkernel/`.

A Example: Atomic Broadcast

A.1 Algorithm

Consider the AB (atomic broadcast) service that uses a simple atomic broadcast algorithm for static groups, defined by the following five rules.

1 To broadcast a message m, process a_i sends the message $[a_i\ t_m\ m]$ to every process (including itself) where t_m is the timestamp of the message, which equals the (logical) time at which the message is sent.

2 When process a_j receives the message $[a_i\ t_m\ m]$ from process a_i, it places it in a local list of messages L, which is ordered by the relation $\Rightarrow$ (defined below). It then sends a (timestamped) acknowledgment message to all processes and tests if a message at the head of L can be delivered to the application (see Step 5).

3 When process a_i receives an acknowledgment message $[a_j\ t_{ack}\ ack]$ from a_j it tests if a message at the head of list L can be delivered to the application (Step 5).

4 Clock Advancing (1) Each process a_i increments its clock C_i between any two successive events, where an event in a process is sending or receiving a message. (2) Upon receiving a message timestamped t_m, process a_i sets its clock C_i greater than or equal to its present value and greater than t_m. (In (2) we consider the event which represents the receipt of the message m to occur after the setting of C_i.)

5 Local Delivery Test Process a_i can remove a message $[a_j\ t_m\ m]$ at the head of list L and deliver it to the application if a_i has received an acknowledment (or other message) from all processes timestamped t_m or later than t_m. In order to implement this condition each process maintains a map M with the timestamps of all processes (initially equal 0); the map is updated each time a new message arrives.

We define a relation $\Rightarrow$ as follows: if m is a message sent by process a_i and m' is a message sent by process a_j, then $m \Rightarrow m'$ if and only if either (i) $t_m < t'_m$ or (ii) $t_m = t'_m \wedge a_i \prec a_j$, where t_m is the (logical) time at which the message m is sent and $\prec$ is any arbitrary total ordering of the processes (e.g. $a_i \prec a_j$ if $i < j$).

The algorithm assumes that for any two processes p_i and p_j, the messages sent from p_i to p_j are received in the same order as they are sent, and every message is eventually received.

A.2 Decomposition

Our goal is to decompose the atomic broadcast algorithm into smaller parts in such a way that some parts could be used in a different context, e.g. to implement another service. We can identify at least four "microprotocols" in the algorithm above, e.g. for dealing with the logical clock, sending a message to a group of processes, deciding about the algorithm's progress on message delivery, and interfacing with the application. The set of events or messages communicated by the microprotocols can be derived from the rules 1-3 above.

To build our example application which provides two services AB and AM (atomic multicast), we have implemented the "microprotocols" as modules called *App*, *LampCast*, *SkeenCast*, *Clock*, and *GroupSend*, where *App* (omitted here) implements a *producer-consumer* algorithm to interface with the application, *LampCast* and *SkeenCast* implement distinct parts of services AB and AM (below we only describe the former one which has been derived from the atomic broadcast algorithm described above), *Clock* implements the logical clock, and *GroupSend* multicasts messages to a group of peers using a lower-level protocol *ReliableP2P* (see Figure 1). The *ReliableP2P* module implements reliable first-in-first-out point-to-point channels above TCP (in Cactus) or UDP (in Appia).

We have used the same code of *App*, *Clock* and *GroupSend* in the implementation of both services. In Appia, we create a new session of *Clock* for each service (see sessions 1 and 2 in Figure 1b), so that each service uses a separate clock counter, however, both services share the same sessions of protocols *App*, *GroupSend*, and *ReliableP2P*. In Cactus, sessions of the composite protocols *LampCast* and *SkeenCast* (implementing the core part of the two services) communicate with one application session and create new sessions of *ReliableP2P* for each new network connection.

A.3 Implementation in Appia

We use module names to denote names of layers and sessions, where session names are marked with a sequence number, e.g. 1,2, to denote many sessions of the same protocol. The protocol stack consists of two communication channels

$$\begin{aligned} \mathit{LampChan} &:= \{\mathit{App}_1, \mathit{LampCast}_1, \mathit{Clock}_1, \mathit{GroupSend}_1, \mathit{ReliableP2P}_1\} \\ \mathit{SkeenChan} &:= \{\mathit{App}_1, \mathit{SkeenCast}_1, \mathit{Clock}_2, \mathit{GroupSend}_1, \mathit{ReliableP2P}_1\} \end{aligned}$$

where an application can use channel *LampChan* to broadcast messages to all processes preserving total order of message delivery, and *SkeenChan* to multicast messages to a group of processes, which is specified by the application, so that the message is delivered to all processes in total order with any other messages that have been sent to any of these processes using this channel type. However, no ordering is done with respect to messages sent using different channel types.

Below, we describe pseudocode for some process a_i that broadcasts messages in a group of processes with identifiers in A. Each process uses two local events: *TimeEvent* and *ClockEvent*; and three sendable events: *AppMsg*, *GroupMsg*, and *AckMsg*. The format of Appia events has been described in §5.4. For each event (T_e, ID, r, n), we denote a routing path r to be either $\uparrow = R[(ID, T_e)]$, or $\downarrow = \mathit{reverse}(R[(ID, T_e)])$, where R has been defined in §5.3. Events are passed using an operation go, either downward ($\downarrow$), or upward ($\uparrow$); the semantics of go is in §5.5.

A.3.1 Module *LampCast*

It implements a unique part of AB, i.e. inserting a message received from below into a list of messages L, sending an acknowledgment message, and testing if a message at the head of list L can be delivered to the application.

The protocol accepts three types of events: *AppMsg*, *AckMsg*, and *TimeEvent*; and generates one (i.e. *AckMsg*). Events *AppMsg* and *AckMsg* are declared as subtypes (denoted :>) of event *GroupMsg*, which is again a subtype of Appia event *Msg* carrying a message.

The protocol uses an auxiliary variable t_{msg} to store a timestamp value of an incoming message. The timestamp is delivered from below by event *TimeEvent*, which is preceding the actual receipt of the message by protocol *LampCast* (see §6 for some explanation why it seems necessary to have a fresh event for this).

Upon receipt of $(\mathit{TimeEvent}, \mathit{LampChan}, \uparrow, [t_m])$
$t_{msg} := t_m$

Upon receipt of $(\mathit{AppMsg}, \mathit{LampChan}, \downarrow, [s{=}\circ\ d{=}A] + m))$
$\mathtt{go}(\mathit{GroupMsg} :> \mathit{AppMsg}, \mathit{LampChan}, \downarrow, [s{=}a_i\ d{=}A] + m)$

Upon receipt of $(\mathit{AppMsg}, \mathit{LampChan}, \uparrow, [s{=}a_{src}\ d{=}A] + m)$
$M := M \oplus (a_{src} \mapsto t_{msg})$
$L := \mathit{sort}_{\Rightarrow}([a_{src}\ t_{msg}\ m] :: L)$
$\mathtt{go}(\mathit{Msg} :> \mathit{AckMsg}, \mathit{LampChan}, \downarrow, [s{=}a_i\ d{=}A])$
$\mathit{DeliveryTest}(M, L)$

Upon receipt of $(\mathit{AckMsg}, \mathit{LampChan}, \uparrow, [s{=}a_{src}\ d{=}A])$
$M := M \oplus (a_{src} \mapsto t_{msg})$
$\mathit{DeliveryTest}(M, L)$

Here is the *DeliveryTest* procedure

Delivery Test (M, L)
$L = [a_j\ t_n\ n] :: L'$
`if` $\forall a_i\ M[a_i] \geq t_n$ `then`
  $L := L'$
  $\mathtt{go}(\mathit{AppMsg}, \mathit{LampChan}, \uparrow, [s{=}a_j\ d{=}A] + n])$

A.3.2 Module *Clock*

It implements a logical clock and timestamps each outgoing message with the current clock value. It accepts one event of type *Msg* (and, of course, all subtypes of *Msg*), and generates two (local) events of type *TimeEvent* and *ClockEvent*.

When a message is received from below on a channel $Chan \in \{LampChan, SkeenChan\}$, the local events *TimeEvent* and *ClockEvent* are created in order to propagate to other layers (on channel *Chan*) a timestamp of the message (used by AB) and the current clock value (use by AM). The event *TimeEvent* is required since the message header which contains the timestamp is stripped by this layer. Notice that if $Chan = LampChan$ then the event *ClockEvent* will be silently discarded by the system since type *ClockEvent* is not accepted by any of the protocols that use channel *LampChan*.

Upon receipt of $(Msg, Chan, \downarrow, attrs + m)$
$C_i := C_i + 1$
$\mathsf{go}(Msg, Chan, \downarrow, attrs + C_i :: m])$

Upon receipt of $(Msg, Chan, \uparrow, attrs + t_m :: m)$
$C_i := max(t_m, C_i) + 1$
$\mathsf{go}(TimeEvent, Chan, \uparrow, [t_m])$
$\mathsf{go}(ClockEvent, Chan, \uparrow, [C_i])$
$\mathsf{go}(Msg, Chan, \uparrow, attrs + m)$

A.3.3 Module *GroupSend*

It multicasts a message to a group of processes A, i.e. for each process $a_j \in A$ it converts name a_j to the process IP address (by looking up the address in a map IP), and passes the message to some (standard) lower-level protocols which implement reliable first-in-first-out communication between machines. The protocol accepts and generates an event type *GroupMsg*.

Upon receipt of $(GroupMsg, LampChan, \downarrow, [s{=}a_i\ d{=}A] + n)$
$\forall a_j \in A\ \mathsf{go}(Msg, LampChan, \downarrow, [s{=}IP[a_i]\ d{=}IP[a_j]] + a_i :: A :: n)$

Upon receipt of $(Msg, LampChan, \uparrow, [s{=}\circ\ d{=}\circ] + a_{src} :: A :: n)$
$\mathsf{go}(Msg :> GroupMsg, LampChan, \uparrow, [s{=}a_{src}\ d{=}A] + n)$

A.4 Implementation in Cactus

We have used the advantage of the Cactus two-level architecture and built service AB by composing "microprotocols" *LampCast*, *Clock*, and *GroupSend* into one composite protocol (also called *LampCast*). This composite protocol forms a protocol stack together with a layer above, which provides an interface with the application, and a layer below that provides an interface with a network protocol TCP (see Figure 1a). The source code of *Clock* and *GroupSend* has been used unchanged in the implementation of a composite protocol *SkeenCast* (not described here) that implements an atomic multicast algorithm of the second service (i.e. AM).

Below, we describe pseudocode for some process a_i that broadcasts messages to a group of processes with identifiers in A. We represent a message as a pair of message attributes and the message type $T \in \{\uparrow, \downarrow, \diamond\}$ (see §4.3 for details). The send votes V^m of Cactus message format are omitted here since in our example only one microprotocol at a time decides about when a message should leave a composite protocol. The operations raise, sendDown, and sendUp, which are used by microprotocols for raising events and sending messages up and down in the protocol stack, are described in §4.2,4.3.

A.4.1 Module *LampCast*

It implements a unique part of AB, i.e. inserting a message received from below into a list of messages M, sending an acknowledgment message, and testing if a message at the head of list M can be delivered to the application.

The microprotocol handles two predefined events carrying a message: *MsgFromUser* and *MsgFromNet*. Each message m has attributes denoted by a record $[t_m\ a_{src}\ A\ msg]$, where attribute t_m contains a timestamp value, a_{src} is the identifier of a processes which sent the message, A is used for outgoing messages and contains identifiers of all processes to which the message is to be broadcast, and msg is the message content. The attributes t_m, a_i, and msg are transmissible attributes (as defined by their visibility scope which is omitted here) and so their values will be used to construct the message header when the message is marshalled into the network format.

Upon event occurrence $(\mathit{MsgFromUser}, m = ([\circ\ \circ\ A\ msg], \downarrow))$
$m := ([\circ\ a_i\ A\ msg], \downarrow)$
raise$(\mathit{AddTimestamp}, \textsc{Sync}, m)$
raise$(\mathit{Multicast}, \textsc{Sync}, m)$

Upon event occurrence $(\mathit{MsgFromNet}, m = ([t_m\ a_{src}\ A\ msg], \uparrow))$
raise$(\mathit{UpdateClock}, \textsc{Sync}, m)$
$M := M \oplus (a_{src} \mapsto t_m)$
`if` $msg \neq Ack$ `then`
  $L := sort_{\Rightarrow}([a_{src}\ t_m\ msg] :: L)$
  $ack := ([\circ\ a_i\ A\ Ack], \downarrow)$
  raise$(\mathit{AddTimestamp}, \textsc{Sync}, ack)$
  raise$(\mathit{Multicast}, \textsc{Sync}, ack)$
$\mathit{DeliveryTest}(M, L)$

Here is the *DeliveryTest* procedure

Delivery Test (M, L)
$L = [a_j\ t_n\ n] :: L'$
`if` $\forall a_i\ M[a_i] \geq t_n$ `then`
  $L := L'$
  raise$(\mathit{Deliver}, \textsc{Sync}, ([t_n\ a_j\ A\ n], \circ))$

A.4.2 Module *Clock*

It implements a logical clock and timestamps each outgoing message with the current clock value. The microprotocol handles two events: *AddTimestamp* and *UpdateClock*. When a message is about to be sent down, the *AddTimestamp* event is raised which initiates a timestamp attribute of the message with the current clock value incremented by one. When a message is received from below the *UpdateClock* event is raised which causes the clock to be advanced.

Upon event occurrence $(AddTimestamp, m = ([\circ\ a_i\ A\ msg], T))$
$C_i := C_i + 1$
$m := ([C_i\ a_i\ A\ msg], T)$

Upon event occurrence $(UpdateClock, m = ([t_m\ \circ\ \circ\ \circ], T))$
$C_i := max(t_m, C_i) + 1$

A.4.3 Module *GroupSend*

The protocol handles two events: *Multicast* and *Deliver*. The handler of event *Deliver* forwards a message to the application layer. The former event causes a message to be broadcast to a group of processes A, i.e. for each process $a_j \in A$ the message is passed to a lower-level session of protocol *ReliableP2P* that maintains a network connection with a_j and can send the message; the bindings of process identifiers a_j to the lower-level sessions are stored in map S. The *GroupSend* protocol opens a new session of *ReliableP2P* dynamically when required, and modifies map S with the new session's name. The protocol *ReliableP2P* resolves abstract process identifiers to IP addresses and communicates messages using a reliable first-in-first-out protocol (TCP).

Upon event occurrence $(Multicast, m = ([t_m\ a_i\ A\ msg], T))$
$\forall a_j \in A$ sendDown$(S[a_j], m)$

Upon event occurrence $(Deliver, m)$
sendUp(m)

We changed the default implementation of sendDown and sendUp, so that the execution of sendDown(m) invokes raise$(MsgFromUser, \textsc{Sync}, m)$ of the lower-level protocol *ReliableP2P*, i.e. the event *MsgFromUser* is raised *synchronously* in order to provide the first-in-first-out guarantee on message delivery required by the atomic broadcast and multicast algorithms. Analogously, the execution of sendUp invokes *synchronously* raise$(MsgFromNet, \textsc{Sync}, m)$ of the higher-level protocol *App*.

B Syntax

B.1 Sets

We use $\mathcal{S}(\mathcal{T})$ to denote all possible subsets of $\mathcal{T}$ (i.e. $\mathcal{S}(\mathcal{T})$ is the powerset of $\mathcal{T}$, denoted $2^{\mathcal{T}}$), and $\mathcal{L}(\mathcal{T})$ to denote all possible lists of elements such that each element is in $\mathcal{T}$.

B.2 Lists

A list of elements in $\mathcal{T}$ is defined in a usual way — as an ordered sequence of elements, together with a concatenation symbol ::, which is used either to append a new element to a list (either as a first, or last element), or in the pattern expression to bind the first element(s) of a list and its tail to local name(s). We also use an operation $setOf(l)$ which returns a set of all elements from list l, $reverse(l)$ which returns a reversed list l, $sort_r(l)$ which returns l sorted by relation r, and $next(e, l)$ which returns the first element following the first occurrence of e in list l (or *null* if e is the last element in l).

B.3 Maps

We represent maps using a set $\mathcal{M}(\mathcal{T} \mapsto \mathcal{T}')$ of all mappings from elements in $\mathcal{T}$ to elements in $\mathcal{T}'$[1], together with following operations

$$
\begin{aligned}
\oplus &: \mathcal{M}(\mathcal{T} \mapsto \mathcal{T}') \times \mathcal{T} \times \mathcal{T}' \rightarrow \mathcal{M}(\mathcal{T} \mapsto \mathcal{T}') \\
\ominus &: \mathcal{M}(\mathcal{T} \mapsto \mathcal{T}') \times \mathcal{T} \rightarrow \mathcal{M}(\mathcal{T} \mapsto \mathcal{T}') \\
[\,] &: \mathcal{M}(\mathcal{T} \mapsto \mathcal{T}') \times \mathcal{T} \rightarrow \mathcal{T}'
\end{aligned}
$$

where $m \oplus (a \mapsto s)$ returns map m with a new binding of a to s (if a was already bound in m, its previous binding disappears), $m \ominus a$ returns map m without a binding of a, $m[a]$ looks up a in map m and returns an element bound to a (if no such binding exists then either a *null* element, or an empty set $\{\}$ is returned). A single mapping from a to s is denoted $(a \mapsto s)$, we use $\circ$ to represent any arbitrary element (thus $(\circ \mapsto a)$ means any binding to a).

B.4 Relations

We use relation $\mathcal{R}$ from a set $\mathcal{T}$ to a set $\mathcal{T}'$, defined as a subset of their Cartesian product, denoted $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}'$ (or $\mathcal{R} \subseteq \mathcal{T}$ if $\mathcal{R}$ is an unary relation, such as *well-formed*); two elements a and b which are in relation $\mathcal{R}$ are denoted $a\ \mathcal{R}\ b$, or $(a, b)\ \mathcal{R}$ (or $a\ \mathcal{R}$ if $\mathcal{R}$ is unary).

[1] We can also think of $\mathcal{T}$ as a type (or class), $\mathcal{S}(\mathcal{T})$ as a bag of values of type $\mathcal{T}$, $\mathcal{L}(\mathcal{T})$ as a type of lists of values of type $\mathcal{T}$, and $\mathcal{M}(\mathcal{T} \mapsto \mathcal{T}')$ as a type of maps storing bindings from keywords of type $\mathcal{T}$ to values of type $\mathcal{T}'$.

B.5 Transition Relation and Operations

We use a transition relation of the form $S, p \vdash op(n) \triangleright S'$, which means that the execution of operation op initiated or invoked by p in some state (or context) S leads to state S'; op has parameters n. Alternatively, we could use notation $op(n) : (S, p) \to (S', p)$. The state is represented by relevant set(s) of elements. In our case, the context of every transition relation is always a single protocol stack, i.e. S always describes (part of) the state of a local runtime system only.

If an operation $op(n)$ is not executed on behalf of the caller but some other principal q then we write $q.op(n)$. If an operation generates some value, we can bind this value to a fresh name, as in $s := q.op(n)$. Thus, we can think of operations as functions (though usually we ignore the result); note that relations, e.g. $c = op$, are functions that return *false* or *true*.

The definition D of op invoked by p is denoted $p \vdash \boxed{op(n):} D$. Each occurrence of op can be replaced by op's definition D using the following expansion. If op is defined as below

$$p \vdash \boxed{op(n):}$$
$$\frac{P(n)}{R, q \vdash op_1 \triangleright R'}$$

then the rule

$$\frac{}{S, p \vdash op(a) \triangleright S'}$$

can be expanded to

$$\frac{P(a)}{S, R, q \vdash op_1 \triangleright S', R'}$$

If an operation op requires some operation op_1 to be executed first (and may also accept results returned by op_1 as arguments), we write

$$\frac{}{S \vdash op(op_1) \triangleright S'}$$

We assume that op is executed *after* op_1 has finished and S is the initial state for *both* operations, i.e. the rule above is an abbreviation for the rule as below.

$$\frac{S \vdash op_1 \triangleright S''}{S'' \vdash op \triangleright S'} \ .$$