A Semantic Model for Adaptive Communication

Sebastian Gutierrez-Nolasco¹, Nalini Venkatasubramanian¹, and Carolyn Talcott²

¹ University of California, Irvine, Irvine, CA 92697-3430 USA
seguti@ics.uci.edu, nalini@ics.uci.edu

² Stanford University, Stanford, CA 94305 USA
clt@cs.stanford.edu

Abstract. Future applications will execute in highly dynamic environments, where communication is not uniform due to changing conditions of network connectivity, reliability and resource availability. Designing communication frameworks that are capable of dynamically adapting themselves to the environment is an important challenge. Furthermore, different applications may demand different properties and guarantees as far as the QoS, security and failure semantics are concerned. Although several frameworks have been designed to support modular and reconfigurable communication protocols, they support complex programming interfaces and protocol behavior is often expressed via informal descriptions. Clear semantic models capable of expressing dependencies among properties required by a given communication system will help design safe customizable communication frameworks. In this paper we present a reflective communication framework based on a semantic model of distributed object reflection. We illustrate how such an adaptive framework is used to formalize and reason about interactions between communication protocols and ensure their safe composition.

1 Introduction

Recent advances in component based software and commercial off-the-shelf products have enabled the creation of applications capable of dynamically adapting themselves to unpredictable changes in the computational and communication environment. However, these new kinds of application may demand diverse guarantees from the underlying communication subsystem as far as QoS (quality of service), security and failure semantics are concerned. Traditional communication frameworks are not well suited to capture and express dependencies among properties required by the application.

For example, today many workers access (via VPNs) confidential business data from almost any wireless hot-spot location such as coffee houses, food-franchises, libraries and conference halls. Currently, these hot-spots offer free network access, but we foresee a shift in this altruistic vision allowing free restricted access; while providing increased bandwidth or greater access to paid customers. In this scenario, a paid customer will expect a certain minimum level of QoS in the form of sustained bandwidth, end-to-end delay bound and a certain degree of security (even if he does not trust the hot-spot infrastructure and uses a VPN, since user authentication is done before obtaining a secure channel or a VPN connection). However, current security protocols assume reliable communication between end-points and all communication (at the application level) is stalled while both end-points authenticate themselves and establish the connection or when they execute refresh or rekey protocols. Therefore, they are not immediately suitable for use in hot-spots, which are characterized by sudden changes in latency, bandwidth and reliability due to unexpected increase of customers in the vicinity or the use of intensive applications (e.g.multimedia, file transfers, etc) by customers on the move. Next generation of communication frameworks must monitor their environment in order to adapt themselves to sudden changes on reliability, latency, bandwidth and security levels provided by the hot-spot without disrupting communication at the application level.

As another example, an unmanned air vehicle (UAV) records surveillance video data and transmits it via a wireless channel to a receiver on a nearby ship. The received video stream must be distributed to several viewers on that ship and possibly on other ships. In this envisioned scenario, data in transfer must be protected from being read by unauthorized entities and compressed in order to overcome bandwidth constraints. Moreover, in a battlefield situation the ship requires timeliness of delivery of the UAV pictures to assure that decisions are made from timely data. Encryption and compression are inherently data intensive and their use affects the timeliness requirement. To further complicate the scenario, depending on their current location, viewers may have different and constantly changing requirements in terms of security and timeliness.

Although these simple scenarios highlight the relevance of adaptive communication, they also underscore the requirement for a semantic model capable of assuring safe adaptation.
In this paper, we describe a reflective communication framework (RCF) and develop a detailed formal semantics for such framework using a generic semantic model of distributed object reflection. We then illustrate how the semantic specification of the RCF is used to ensure correctness of critical application properties and how it can be used to support adaptive communication. The remainder of this paper is organized as follows. In Section 2, we describe the basic reflective communication framework. Formal specification of the framework is discussed in Section 3. Section 4 gives an overview of the formal semantics of the reflective communication framework; while Section 5 discusses composability and interaction issues that arise in adaptive communication. We describe related work and conclude in Section 6 with future research directions.

2 A Reflective Communication Framework

Since the actor model of computation [7] incorporates the notion of encapsulation and interaction only via message passing, it offers a clear, flexible and simple semantic approach to describe distributed systems based on incoming communications [9]. The system is modeled as a group of self contained and independent autonomous objects, called actors, which communicate via asynchronous (buffered) message passing. On receiving a communication, an actor processes the message in a manner determined by its current behavior. As a result, the actor may: (1) Create new actors, (2) Change its behavior and (3) Send messages to itself or to other (existing) actors. Since mail addresses may be communicated in messages the configuration of the communication is dynamic and the activation order (one message activates another if the latter is sent during the processing of the former) determines communication patterns.

In our model, a system is composed of two kinds of actors, base actors and meta actors, distributed over a network of processing nodes. Base actors carry out application level computation, while meta actors are part of the run-time system, which manages system resources and controls the run-time behavior of the base level. Meta actors communicate with each other via message passing as do base actors, and they may also examine and modify the state of the base actors located on the same node. Base level actors and messages have associated run-time annotations that can be set and read by meta actors. The annotations are invisible to base level computation. Actions which result in a change of base level state, are called events and meta actors may react to them if they occur on their node. A configuration has a set of base and meta level actors and a set of undelivered messages. The actors are distributed over the system nodes. Each actor has a unique name (actor identifier) and the configuration associates a current state to each actor name. The undelivered messages are distributed over the network (some are traveling along communication links and others are held in node buffers).

In order to provide correct composition of communication protocols in a transparent and scalable fashion, we extend our model with a reflective communication framework (RCF), which customizes the base level communication as follows (see Figure 1). Each base level actor has a meta level actor, called messenger, which serves as the customized and transparent mail queue for that base level actor. There is one communication manager in every node of the distributed system, which implements and controls the correct composition of communication protocols specified by messengers on that node. A messenger has two message queues: (i) The up queue is used as the (base level) actor’s send buffer, and (ii) The down queue is used to deliver messages to the (base level) actor, serving as its customized mail queue. The communication manager has a set of communication protocol actors, each of them implementing a particular communication protocol provided by the framework (e.g. reliable, in-order or security). This scheme allows us to abstract a core set of communication protocols and share it between the different messengers on a node, simplifying the synchronization and composition process. Furthermore, it encourages separation of concerns in the process of message transmission and reception. However, in purely reflective architectures, reasoning about the semantics of correct communication composition may be complicated; moreover, its implementation may be inefficient.

In order to maintain accurate semantics and provide an efficient implementation of the architecture, the communication manager uses a pre-defined pool of meta level entities, called communication message coordinators (or simply message coordinators). Every message requiring a communication protocol is assigned a message coordinator and at any instance, a message coordinator handles the composition of the communication protocols requested by a messenger for an individual message. The message coordinator assures the correct order of composition of required protocols and provides a coordination mechanism between the messenger and the protocols that provide it. This concept of reusable message coordinators is an efficient way to handle the service request of each messenger without having to pay the bottleneck associated with the centralization of the services in the node communication manager and allows us to process concurrently multiple messages.

In the RCF model, a communication protocol may be explicitly requested (at any instance) by the sender or implicitly specified by its messenger, in case the sender and the receiver previously agreed to use a particular communication
In order to provide dynamic customization of communication protocols (potentially on a per message basis), the RCF model separates specification, composition and implementation of communication protocols. The specification is handled by the messenger through a message service list (msl), which determines the set of communication protocols to be used in the communication of a specific message to its target. The communication protocols required by a communication may be explicitly specified or constructed by the messenger using target-actor information within the message. The messenger also assigns every outgoing message a unique message identifier (msgid), which is a unique sequence of values assigned to every message to be sent. The msgid is composed by concatenating the communication manager identifier (cmid), an application identifier (appid), a communication manager sequence number (cms) and a local time stamp (ts) defined by Lamport’s happened-before relation [16]. The protocols themselves are implemented by independent communication protocol actors. This scheme allows us to add (plug in) or remove (plug out) communication protocols dynamically.

3 Russian Dolls: A semantic model of distributed object reflection

The concurrent state of a distributed system, often called a configuration, has typically the structure of a flat multiset made up of objects and messages that behave according to a set of rules describing the behavior of individual objects. We can visualize this configuration as a soup in which objects and messages float and interact with each other. We model a distributed system configuration using the rewriting logic object model [26] [13] where an object in a given state is represented as

\[ <O:F|atts> \]

Where \( O \) is the object’s name or identifier, \( F \) is its class and \( atts \) its attribute set, and the rewrite rules describing the dynamics of the system can have the form

\[ r : M_1...M_n, <O_1:F_1|atts_1>...<O_n:F_n|atts_n> \]

\[ \Rightarrow <O_1:F_1|atts_1>...<O_n:F_n|atts_n> ... <O_i'* ...'F_i|atts_i>' ... <O_n'* ...'F_n|atts_n'> M_1''''...M_n'''' \text{ if } C \]

where \( r \) is the label, \( M \)'s are messages, and \( C \) is the rule condition. That is, at any time, objects and messages can come together and participate in a concurrent transition corresponding to a communication event in which some new objects may be created (\( O_{\text{new}} \)), other objects may be destroyed, others may change their state and new messages (\( M_{\text{new}} \)) may be created.
3.1 Components of the Model

In the Russian Dolls model [14], objects are structured in nested configurations of meta objects that can control (base and meta) objects under them, called subobjects, with an arbitrary number of levels of nesting. This allows us to model processing nodes as meta objects that encapsulate actors (base and meta) and messages. Thus, we have three main components in the model: actors (base and meta), nodes and messages.

**Base and meta actors:** We use the Russian Dolls theory to model actors as objects with identity and state, configurations as soups of actor and messages, and actor behavior rules as rewrite rules. Therefore, a base actor system is represented as a special case of object module configuration. A base actor is represented as an object in some base actor class $B_C$. Messages have the form $a \langle m \rangle$, where $a$ is the actor identifier of the target actor and $m$ is the message contents. Base actor rules are constrained to have the form

$$<a \langle BC | atts \rangle , [a\text{m}] \Rightarrow <a \langle BC' | atts' \rangle , bconf \text{ if cond}>$$

where $BC$ and $BC'$ are base classes, $atts$ and $atts'$ are attribute value sets appropriate for the corresponding classes, $bconf$ is a configuration of newly created base actors and messages, and $[\ ]$ indicates that the message part is optional. A meta actor is an object in some meta actor class $MC$ and meta messages have the form $ma <mv$, where $ma$ is the identifier of the meta actor. Meta actors are grouped as subobjects configurations of nodes.

**Node:** A node is an object of class $Node$ and has an attribute $conf$ whose value is a configuration containing meta actors and messages. Each node has two special meta actors (see figure 2): A base behavior task manager meta actor $btma$ of class $BTMC$ that represents and controls base execution behavior, and a communication meta actor $cma$ of class $CMC$ that represents and controls base communication semantics. These meta actors also implement the event handling semantics associated with base events. Hence, a node has the form

$$<v:Node | conf:<btma:BTMC|...> <cma:CMC|...> mconf>$$

![Fig. 2. The Russian Dolls Model](image)

**Base behavior task manager actor:** A base behavior task manager meta object of class $BTMC$ has attributes $bMod, bConf, eReg, pendEv, waitFor$.

$$<btma:BTMC | bMod | BM, bConf | bc, eReg | emap, pendEv | ev, waitFor | W>$$

$BM$ is the module that describes the behavior of the base level object as rewrite rules and thus, the value of $bMod$ is the meta representation of the base module $BM$; while the value of $bConf$ is the meta representation of the base configuration $bc$. Here we use the convention that if $abc$ denotes a base entity then $labc$ denotes the meta representation of that entity. The fact that rewriting logic is reflective guarantees that $labc$ is the meta representation of $abc$ and such faithful meta representation exists.
value for an event only if it has an event handling rule matching this event that generates a reply to the notifying actor. The notified about the event. The mapping is computed using the term base actors and messages to identifiers of existing base actors on whose behalf they were created.

Event handling is modeled by sending notifications to meta actors registered for the event. A meta actor is registered for an event only if it has an event handling rule matching this event that generates a reply to the notifying actor. The value $emap$ of the attribute $eReg$ maps an execution event to the set of names of meta actors that are registered to be notified about the event. The mapping is computed using the term $apply(emap, event)$. The value $ev$ of the attribute $pendEv$ is either an execution event or a special value $none$, indicating that no event handling is in progress. The value $W$ of $waitFor$ is the set of meta actor identifiers for which the notification reply has not been received.

**Communication actor** A communication meta object of class $CMC$ has attributes $sendQ$, $arriveQ$, $eReg$, $pendEv$, $waitFor$. The value of $sendQ$ is a list of outgoing messages of the form $(msg, sndr)$, representing requests from base actor $snidr$ to send message $msg$; whereas the values of $arriveQ$ is a list of incoming messages of the form $(msg, lsnidr, amap)$ representing the arrival of message $msg$ from base actor $snidr$ with message annotation map $amap$. The remaining attributes are event handling attributes similar to those of $BTMC$.

$$<cma: CMC|sendQ:msgs, arriveQ:msgs, eReg:emap, pendEv:ev, waitFor:W>$$

### 3.2 Meta level transitions

Since a system configuration can be visualized as a soup of nodes and messages with base and meta actors forming subgroups inside the nodes, a node transition might be either a communication transition or an execution transition.

A communication transition moves a message to actors on other nodes from the node’s buffer to the external soup or moves a message to an actor on the node from the external soup into the node’s mail buffer. An execution transition first applies either a base or a meta transition rule. If there is an associated base event, then each event handling rule that matches the event must be applied (in an unspecified order) and the associated annotation update applied to the new local base configuration and messages. In general, a meta transition rule has the form

$$\text{ma [msg]} \rightarrow^{\text{check f}} \text{ma', mconf if cond}$$

where $ma$ is a meta actor, $[msg]$ is an optional meta message addressed to $ma$, $ma'$ is the same meta actor with its state possibly modified, $mconf$ is a configuration of newly created base actors and messages and $cond$ is a predicate constraining the actor-message pairs to which the rule applies. $update$ is either empty or has the form $(bconf, cmap)$. When the rule is applied, $bconf$ is bound to the base actor configuration on the node where the meta actor is located. As a result of rule application, base actor annotations or message annotations may be updated. Meta actor rules with empty update are represented directly as object rewrite rules

$$<\text{ma:MC|atts}> [msg] \rightarrow <\text{ma:MC'}|atts'>, mconf if cond$$

Rules with non-empty update require synchronization with the $btma$ since the $btma$ needs to notify all registered meta actors before updating the base configuration. In order to clearly specify base level execution, we divide the execution of a base actor into four consecutive steps or rewrite rules, each of them marked by a label of the form $RR(label)$. (see figure 3). In each step, some attributes are modified and their modification will enable the next step to execute. That is, initially the $btma$ will enqueue the execution event in its $pendMsg$ attribute (step 1), enabling step 2 to proceed, which notifies all registered meta actor that the event $evt$ is going to be executed. Note that $notifyall(W, evt)$ is the set of messages $ma < notify(fy(evt))$ for $ma$ in $W$. That is, $W$ is the set of meta actors registered to be notified when the event $evt$ occurs. Once the notifications have been sent, the $btma$ starts processing the event (step 3), which may produce new actors and messages. $cmap$ maps new actors and messages to actor $a$. That is, newly actors and messages were created on behalf of actor $a$. When the execution of the event is being processed (step 4), the $btma$ waits for acknowledgements from the registered $ma$‘s. When all registered meta actor have acknowledged notification, $btma$ completes the execution event by updating its model of the base configuration and transmitting newly sent
messages to the $cma$ (step 5). $bc'$ is the result of applying $update$ to $b_c$, and $smgs'$ contains elements of the form $(!msg, cmap[msg])$ for each message specified by $update$

<table>
<thead>
<tr>
<th>Step 1</th>
<th>$R^{def}(InitMsgEvent)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : none, started : false &gt;$</td>
</tr>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : ext, started : false &gt;$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>$R^{def}(startMsgEvent)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : ext, started : false &gt;$</td>
</tr>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : ext, started : true, waitFor : W &gt;$, notifyall($W, ext$) where $W = apply(emap, ext)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>$R^{def}(startExecution)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : none, started : true &gt;$</td>
</tr>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : ext, started : false &gt;$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>$R^{def}(continueExecution)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : ext, started : true, waitFor : Wma &gt;$, btmn $\not\in$ notifyall$@bma$</td>
</tr>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : ext, started : true, waitFor : W &gt;$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 5</th>
<th>$R^{def}(completeExecution)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc, eReg : emap, pendEv : exe(!BM</td>
</tr>
<tr>
<td>$&lt; btmn : BTMC</td>
<td>bMod : !BM, bConf : !bc', eReg : emap, pendEv : none, started : true &gt;$, $&lt; cma : CMC</td>
</tr>
</tbody>
</table>

Fig. 3. Execution rules

4 The Extended Russian Dolls Model

The $RCF$ functionality is introduced in the basic Russian dolls model in order to formalize protocol interactions and for stating and proving certain properties that ensure safe protocol composition. Basically, specialized processing of messages are handled by the $RCF$ before they reach the $cma$. The $RCF$ is an instance of class $RCFmodule$ and has an attribute $conf$ whose value is a configuration containing: (i) a communication manager meta actor $\rho$ of class $CMgrC$ that coordinates message processing, and (ii) a set of messenguer meta actors $\sigma$ of class $MsgrC$ that controls message semantics for their base actors. Thus, the $RCF$ has the form

$$<rcf:RCFmodule|conf : <\rho : CMgrC|... > <\sigma : MsgrC|... > mconf>$$

We expand the communication manager to encapsulate and control a set of available communication protocols $\delta$ of class $CPC$ and a pool of message coordinators actors $\rho$ of class $MCC$, so it can coordinate message processing in the node.
Communication Manager: A meta object of class $CMgrC$ has attributes $in$, $out$, $cpConf$, $conf$. The values of $in$ and $out$ are lists of incoming and outgoing messages of the form $(msg*)$, representing processed messages. The value of $cpConf$ is the meta representation of the communication protocols currently loaded; while the value of $conf$ is a configuration containing message coordinators and communication protocols.

$\langle \delta : CMgrC | in : msg, out : msg, cpConf : cp \rangle, \langle \delta : CPC | \rangle, mConf>\$

Message Coordinator: A meta object of class $MCC$ has attributes $mailq$, $mpr$, $mrst$, $mip$, $ord$, $stage$, $spin$. The value of $mailq$ is a list of incoming messages of the form $(msg, sndr, amap)$ representing the request to process message $msg$ sent by base actor $sndr$ and message annotation map $amap$ with the communication protocols specified in $msl$. The values of $mpr$, $mrst$ and $mip$ are meta representations of the master prerequisites, restrictions and interaction parameters (respectively) of the communication protocols specified in $msl$; while the value of $ordl$ is the meta representation of the composition order that needs to be obeyed to ensure correct composition of communication protocols.

$\langle \rho : MCC | mailq : msg, pr : plist, rst : plist, ip : plist, ord : ordl, stage : stage, spin : false \rangle$\$

Communication Protocol: A meta object of class $CPC$ has attributes $sndq$, $rcvq$, $pr$, $rst$, $ip$, $statefull$. The value of $sndq$ is a list of outgoing messages of the form $(msg*)$, representing a processed message to be returned to the corresponding $\rho$ that requested the service; whereas the value of $rcvq$ is a list of incoming messages of the form $(msg, sndr, amap)$ representing the arrival of message $msg$ to be processed by the protocol. The values of $pr$, $rst$ and $ip$ are meta representations of the protocol prerequisites, restrictions and interaction parameters respectively.

$\langle \delta : CPC | sndq : msg, rcvq : msg, pr : plist, rst : plist, ip : plist, statefull : false \rangle$\$

Messenger: A meta object of class $MsgC$ has attributes $up$, $dwn$, $lastmsgid$, $lastmsl$, $msgclass$. The values of $up$ and $dwn$ are lists of incoming and outgoing messages of the form $(msg, sndr, amap)$, representing raw messages. The values of $lastmsgid$ and $lastmsl$ are meta representations of the message identification and message service list respectively.

$\langle \theta : MsgC | up : msg, dwn : msg, lastmsgid : msgid, lastmsl : msl \rangle$\$

Before giving a concrete illustration of the model, we need to formalize the enqueing of a message, we define the enqueing relation $Q_1 \in enQue(Q_0, m)$. $Q_1$ is an enqueing of message $m$ onto $Q_0$ by $Q_1 \in enQue(Q_0, m)$. $Q_1 \Leftrightarrow Q_1 = Q_0 \circ Q_2$ for some sequence $Q_2 \sim m$.

Message Arrival

Let us briefly describe the system behavior during an incoming message when the RCF module is both inactive and active. A message arrival event has the form $arrive((a \triangleq m), msg, amap)$ indicating the arrival of message $m$ for base actor $a$, sent by $sndr$ and having annotations $amap$. On the other side, a message send event has the form $send((a \triangleq m), msg, amap)$ indicating that a message $m$ having annotations $amap$ is going to be sent to $a$ by $sndr$.

Figure 4 shows the communication rules that need to be applied when a message of the form $cma \triangleleft arrive(msg, sndr, amap)$ arrives to the node. Since the RCF is not active by default, the message is queued for later processing into the $cma$ mail queue (step 1). Before the $cma$ can deliver a message, it needs to notify all meta actors registered for message arrival (step 2) and waits for the acknowledgement of all notified meta actors (step 3). Finally, it delivers the message to the $btma$, which will deliver the message to the target actor (step 4). The complete semantics of message arrival using the RCF is described in the Appendix A.

---

1 we keep message annotations at the meta level. In case of base actor annotations, we let the meta actors maintain them internally. Thus, annotations are modified through the meta level
5 Towards Adaptive Communication

In order to respond to sudden changes in the communication environment without disrupting the application, the communication framework must continuously monitor the network environment and adjust, replace or compose communication protocols accordingly. However, specifics of a communication protocol restrict the adjustments that can be made to the protocol itself and may pose conditions (on the system environment) that prevent or constrain its combination or concurrent execution with other protocols. When two or more communication protocols are composed in order to obtain their combined benefits their guarantees (desirable properties) are not always preserved. Preservation of protocol guarantees may crucially depend on the composition order and their interaction parameters, which constrain communication protocol behaviors to achieve their safe composition.

Fig. 4. Message arrival rules

5.1 Case Study: Secure Delivery in Hot-spots

Current security protocols are implemented as a single (ordered) sequence of messages between participants without choice points (ifs) or loops. In fact, they assume reliable transfer between static end-points and stall all communication (at the application level) while the end-points are authenticated and a virtual secure connection is established (or re-established) between them (i.e. the user is watching an hour-glass cursor all this time). As we briefly discussed in Section 1, hot-spots are characterized by sudden network changes and mobile end-points that connect and disconnect frequently. In this scenario, an established mobile-to-mobile end-point session may crosscut several network domains, each one of them with possible different security levels and network characteristics, and session refresh may be required frequently. Thus, instead of freezing all communication every time a refresh is triggered, we would like to execute session refresh (in the background) while (application level) communication is in progress. For example, let us consider the popular SSL/TLS protocol [4] as our secure protocol. Here, we use RSA to authenticate both end-points (the original motivation for SSL and still by far its most common use). We recall that RSA uses nonces\(^2\), a random

\(\text{Step 1 } RR^{\text{arriveq}}\) 
< cma : CMC|arriveQ : amsgs >, cma \& arrive(!msg, !sdn, amap)  
implies  
< cma : CMC|arriveQ : amsg o (!msg, !sdn, amap) >

\(\text{Step 2 } RR^{\text{startArrival}}\) 
< cma : CMC|arriveQ : amsg o (!msg, !sdn, amap), eReg : emap, pendMsg : none >  
implies  
< cma : CMC|arriveQ : amsgs, pendMsg : msg, waitFor : W >, notifyAll(W, event)  
if event == arrive!(msg, !sdn, amap) and W == apply(emap, event)

\(\text{Step 3 } RR^{\text{continueArrival}}\) 
< cma : CMC|arriveQ : amsgs, pendMsg : msg, waitFor : Wma >, cma \& notified(ma)  
implies  
< cma : CMC|arriveQ : amsgs, pendMsg : msg, waitFor : W >  
if (ma  W)

\(\text{Step 4 } RR^{\text{completeArrival}}\) 
< cma : CMC|arriveQ : amsgs, pendMsg : msg, waitFor : mt >,  
< bma : BMC|bMod : BM, bConf : bc, pendMsg : none >  
implies  
< cma : CMC|pendMsg : none >, < bma : BMC|bMod : BM, bConf : delivery(BM, bc, msg) >

It is assumed that these numbers are generated in such a way that they can not be guessed.
number that is used in a single execution of the protocol, and public key cryptography. i.e. each object possesses a public key which can be accessed by all objects and a secret key, which is required to decrypt any message encrypted with its public key. In order to avoid freezing application level communication while the key refresh is in progress, we make a distinction between a connection and a session. A connection represents one specific communication channel; while a session is a virtual construct representing the negotiated algorithm and the master = secret used to create the secret key. Then, we allow multiple connections to be associated to a given session and we switch connections when the new connection is ready. As a result, we can divide the execution of the combined protocols in four phases as follows (see figure 5):

Session setup: A secure communication channel is created by authenticating both the initiator A and the responder B and by establishing the keying material (shared secret and participants’ random values) required to generate a secret key, which is used to protect further application traffic as follows:

1. A creates a nonce NA, adds its address as an identifier and encrypts this information with B’s public key of PKB and sends message M1 to B
2. Upon receiving M1, B decrypts the message using its private key SKB, creates a new nonce NB, generates a new session identifier sid and choose a ciphersuite, which determines the encryption algorithm to be used and sends message M2 back to A, with a concatenation of both nonces, the session identifier sid and the ciphersuite cipher encrypted with A’s public key PKA
3. A receives M2 and decrypts it. In case the contents of M2 is the concatenation of his own nonce (NA) plus another nonce (NB), A generates a pre-master secret and sends it with the separated nonce NB to B as part of message M3. A uses the pre-master secret in order to generate the secret key SKey used to actually encrypt/decrypt further (application) traffic belonging to this session
4. Upon M3 reception, B decrypts it and uses the pre-master secret to generate the secret key SKey

Session established: Once M3 has been received, the session is formally established and any further communication sent or received is encrypted/decrypted using the secret key

Session refresh: If one of the participants suspects eavesdropping, a session refresh may be requested. Refresh replaces (part of) the shared secret used to generate the secret key. In order to avoid freezing application level communication while the key refresh is in progress, we make a distinction between a connection and a session. A connection represents one specific communication channel; while a session is a virtual construct representing the negotiated algorithm and the master = secret used to create the secret key. Then, we allow multiple connections to be associated to a given session and we switch connections when the new connection is ready:

1. A requests a session refresh by sending a message M4 with the current session identifier sid and a new nonce NA as parameters of the refresh request. 4 is encrypted using SKey and it is sent to B
2. Upon receiving M4, B decrypts the message and accepts the request by generating a new nonce NB and sending a message M5 back to A. M5 encapsulates the session identifier and the concatenation of both nonces and it is encrypted by SKey
3. In case the contents of the received and decrypted M5 are the concatenation of A’s nonce (NA) plus another nonce (NB) and the session identifier correspond to the current in progress, A generates a new-pre-master secret and sends it with the separated nonce NB and session identifier sid to B as message M6. Then, A uses the new-pre-master secret to generate a new secret key newSKey
4. When B receives M6 and decrypts it, B uses the new-pre-master secret to generate the new secret key newSKey and sends back an acknowledgement message M7 to A, with the session identifier sid and the finished keyword, indicating that all further communication is processed using the new key
5. Upon receiving M7, A uses newSKey to process previously queued (application level) messages and the key refresh is terminated

Session teardown: When a secure communication is not longer desired or needed, a teardown session request is generated by one of the participants, the connection is removed

---

3 Although all connections in a given session share the same master = secret, each one of them has its own encryption keys.
4 The key does not necessarily need to be a symmetric key
5 Although all connections in a given session share the same master = secret, each one of them has its own encryption keys.
Specifying the Protocols in Our Model

Although both protocols execute correctly in isolation, it is necessary to ensure that they do not interact in harmful ways when they are combined and executed concurrently. In order to achieve this, we use the extended Russian dolls model to specify the combined protocols as follows: We define the class Agent as a generic class and classes Initiator and Responder as its subclasses, one for each possible role of the protocol. Agent have attributes secKey, stage, ecom, counter, session, ciphersuite, skey, queue. The values of secKey and skey are the private and secret keys, ciphersuite stores all the information regarding the algorithm selected; while ecom stores the information about the established communication. sid is the session identifier, stage keeps track of the protocol execution stage, so participants know what message they should expect, and counter is a sequence number, used to generate nonces. The value of queue is a list of incoming messages of the form msg representing the request to process message msg. Initiator and Responder have two private attributes bObId and AliceId respectively. bObId stores the identifiers of the intended Responder; while AliceId stores the identifier of the intended Initiator. As a second step, we specify the execution in rewriting rules, one for each message exchanged between participants (see Figures 6-8). Finally, we integrate them into the extended Russian dolls model as follows (the complete set of rules required are described in the Appendix B). Let us assume that agent A wishes to communicate with agent B in a secure manner. When A sends message m1 to B, m1 is intercepted by its corresponding messenger Msg(A), who requests (on A’s behalf) a secure connection by to its local communication manager Cmgr(A). Cmgr(A) in turn selects a message coordinator to handle the request and interact with the protocols (described above). Once both protocols have been authenticated an a secure session has been established, m1 can be encrypted and send it securely to through the network.

5.2 Reasoning about Safe Composition

Isolated protocol properties, such as confidentiality, data integrity, authentication or secrecy of sessions keys, are worthy of study and have been expressed and proved elsewhere. However, protocol specification and reasoning should
concern not only these intrinsic properties but also their interaction with other protocols (after all, protocols are part of a larger system) and how far can the assumptions be relaxed without compromising protocol correctness.

e.g. Rekey and refresh protocols block communication at the application layer in order to ensure safe key generation and exchange. In this particular scenario, safe means that no application level messages are floating in the network when the key is generated or exchanged, ensuring that every message send with the old key will be received and decrypted before keys are switched. Allowing messages to be send and receive while the refresh protocol is in progress, require us to define a synchronization point where keys can be switched (even if they are messages floating in the network). This implies a way to identify and keep track of every message, so messages sent but not received before the keys were switched can be properly retransmitted using the new key. As a result of removing the synchronization

Since messages may be floating in the network when the switch took place, old messages may arrive after the keys have been switched, and will be discarded as corrupt messages (since there is no way to decrypt them and they have been retransmitted using the new key). We use our model as a framework for stating and proving the following (safety and liveness) properties:

1. Refresh Idempotency: There is only one session refresh in progress between participants at any point of time.
2. No Message Loss due to Session Refresh: All (base-level) messages sent while the session refresh is in progress are eventually delivered to the application.
3. No Message Duplication: No duplicated message is delivered to the application.

---

**Fig. 6.** Session setup and session established rules

---

<table>
<thead>
<tr>
<th>Step</th>
<th>Rule</th>
</tr>
</thead>
</table>
| Step 1 $R^{A}_{init}$(startRun) | $A: \text{Initiator}[\text{secKey}: SKA, bobId: B, stage: 0, ecom: none, counter: J, session: none, cipherSuite: none, skey: none, queue: none]$
\implies
A: \text{Initiator}[\text{secKey}: SKA, bobId: B, stage: 1, ecom: none, counter: J + 1, session: none, cipherSuite: none, skey: none, queue: none],
B < M1(encrypt((nonce(A, J), A), pubKey(B))) |
| Step 2 $R^{A}_{init}$(nonce:Change) | $B: \text{Responder}[\text{secKey}: SKB, aliceId: none, stage: 0, ecom: none, counter: K, session: none, cipherSuite: none, skey: none, queue: none]$
\implies
B: \text{Responder}[\text{secKey}: SKB, aliceId: A, stage: 1, ecom: none, counter: K + 1, session: sid, cipherSuite: cipher, skey: none, queue: none],
A < M2(encrypt(((NA, nonce(B, K)), newSid(A, K), cipher), pubKey(A))) if (keyPair(SKB, PKB) and (sid = newSid(A, K))) |
| Step 3 $R^{A}_{init}$(nonce:Ack) | $A: \text{Initiator}[\text{secKey}: SKA, bobId: B, stage: 1, ecom: none, counter: J + 1, session: none, cipherSuite: cipher, skey: none, queue: none]$
\implies
A: \text{Initiator}[\text{secKey}: SKA, bobId: B, stage: 2, ecom: B, counter: J + 2, session: sid, cipherSuite: cipher, skey: Skey, queue: none],
B < M3(encrypt(((NB, sid, pre − master), pubKey(B))) if (keyPair(SKA, PKA) and (NA == nonce(A, J)) and (sid = newSid(A, K)) and (Skey = makeKey(pre − master, NA, NB))) |
| Step 4 $R^{A}_{init}$(Authenticated) | $B: \text{Responder}[\text{secKey}: SKB, aliceId: A, stage: 1, ecom: none, counter: K + 1, session: sid, cipherSuite: cipher, skey: none, queue: none]$
\implies
B: \text{Responder}[\text{secKey}: SKB, aliceId: A, stage: 2, ecom: A, counter: K + 2, session: sid, cipherSuite: cipher, skey: Skey, queue: none],
if (keyPair(SKB, PKB) and (NB == nonce(B, K)) and (sid = newSid(A, K)) and (Skey = makeKey(pre − master, NA, NB))) |
**Fig. 7.** Session refresh rules

<table>
<thead>
<tr>
<th>Step 5</th>
<th>RR&lt;sup&gt;4&lt;/sup&gt;Is(RefreshReq)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Step 6</th>
<th>RR&lt;sup&gt;4&lt;/sup&gt;Is(RefreshAccepted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;B: Responder</td>
<td>secKey: SKB, aliceId: A, stage: 2, ecom: A, counter: Q, session: sid, ciphersuite: cipher, skey: Skey, queue: M4(encrypt(NA, A, sid), Skey) &gt;</td>
</tr>
<tr>
<td>➞</td>
<td>&lt;B: Responder</td>
</tr>
<tr>
<td>if(NA == aliceId)and(sid == getSid(decrypt(M4, SKey))))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 7</th>
<th>RR&lt;sup&gt;4&lt;/sup&gt;Is(newKey)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;A: Initiator</td>
<td>secKey: SKA, bobId: B, stage: 3, ecom: B, counter: P + 1, session: sid, ciphersuite: cipher, skey: Skey, queue: M5(encrypt((NA, NB, sid), Skey)) &gt;</td>
</tr>
<tr>
<td>➞</td>
<td>&lt;A: Initiator</td>
</tr>
<tr>
<td>if((NA == nonce(A, P))and(sid == getSid(decrypt(M5, SKey))))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 8</th>
<th>RR&lt;sup&gt;4&lt;/sup&gt;Is(Finished)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;B: Responder</td>
<td>secKey: SKB, aliceId: A, stage: 3, ecom: A, counter: Q + 1, session: sid, ciphersuite: cipher, skey: Skey, queue: M6(encrypt((NB, sid, new - pre - master), Skey)) &gt;</td>
</tr>
<tr>
<td>A</td>
<td>M7(encrypt((finished, sid), Skey))</td>
</tr>
<tr>
<td>if((NB == nonce(B, Q))and(sid == getSid(decrypt(M6, SKey))))</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 8.** Session teardown rules

<table>
<thead>
<tr>
<th>Step 9</th>
<th>RR&lt;sup&gt;4&lt;/sup&gt;Is(TearDownReq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;A: Initiator</td>
<td>secKey: SKA, bobId: B, stage: i, ecom: B, counter: R, session: sid, ciphersuite: cipher, skey: Skey, queue: none &gt;,</td>
</tr>
<tr>
<td>➞</td>
<td>&lt;A: Initiator</td>
</tr>
<tr>
<td>B</td>
<td>Mn(encrypt((closed), Skey))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 10</th>
<th>RR&lt;sup&gt;4&lt;/sup&gt;Is(TearDown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;B: Responder</td>
<td>secKey: SKB, aliceId: A, stage: i, ecom: A, counter: S, session: sid, ciphersuite: cipher, skey: Skey, queue: Mn(encrypt((closed), Skey)) &gt;</td>
</tr>
<tr>
<td>➞</td>
<td>&lt;B: Responder</td>
</tr>
</tbody>
</table>

12
4. Refresh Progress: For all possible refresh protocol executions, either a successful refresh is achieved or the refresh request is denied.
5. Refresh Termination: Every refresh protocol execution eventually terminates.

6 Related Work and Concluding Remarks

Commercially available distributed middleware infrastructures have incorporated the notion of reflection in order to provide the desired level of configurability and openness in a controlled manner [5][1][25]. The focus of the work presented in this paper is on developing formal semantics and reasoning for safe composition of middleware services and communication protocols. The OpenOrb reflective architecture [17] [6] [11] is based on the RM-ODP object model and inspired by the AL-1/D reflective programming language for distributed applications [10]. In the OpenOrb architecture every object is represented by multiple models allowing behavior to be described at different levels of abstraction and from different points of view. Similarly, [12] provides a meta level architecture at the language level to support a limited fault tolerant system mechanisms, in particular replication. Two reflective architectures for actor computation have been used as a basis for defining and reasoning about composable services in dynamic adaptable distributed systems: The onion skin model as a basis [15] and the two-level actor machine (TLAM) model [23] [22]. The onion skin model has been used to support high level abstractions such as synchronizers [30], real time synchronizers [29], actor spaces [8], and dynamic architectures [18] [3]; while the TLAM has been used to reason about safe composition of system-level activities, such as distributed garbage collection [24], migration and global snapshot [21], and global snapshot in the presence of failures [28].

References

A Message Arrival using the RCF:

Let us assume that the RCF module is active, then every message may be processed by a set of communication protocols that the \texttt{cma} is not capable to identify or read. As a result, the \texttt{cma} is not able to extract the information it requires to determine the possible set of meta actors that need to be notified for a message arrival. Thus, a message may need to be unwrapped before the \texttt{cma} can notify its arrival. In order to achieve this, we define an interface (see figure 9) between the \texttt{rcf} and the \texttt{cma} such that all incoming messages are forwarded to the \texttt{rcf} as soon as the arrive to be unwrapped (step 1) and all unwrapped messages are returned to the \texttt{cma} for further event notification and delivery (step 2) following the same message arrival rules described in figure 4. We further divide the unwrapping process in three sections: message handling, message processing and message delivery.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Step 1} \textit{RCF}($\texttt{cma}$\texttt{2cmgr}) \\
$\langle \texttt{cma} : \texttt{CMC} | \texttt{arriveQ} : \texttt{amsq} , \texttt{sendQ} : \texttt{smqs} \rangle$ , \texttt{cma} $\triangleleft$ \texttt{arrive}('\texttt{msg}!',\texttt{snr},\texttt{amap}), \\
$\langle q : \texttt{CMgrC} | \texttt{in} : \texttt{imq} , \texttt{out} : \texttt{omq} , \texttt{cpConf} : \texttt{cps} , \texttt{conf} : \langle \rho : \texttt{MCC} \rangle \ldots \rangle , \langle \delta : \texttt{CPC} \rangle \ldots \rangle , \texttt{mconf} \rangle$ \\
\hline
$\implies$ \\
$\langle \texttt{cma} : \texttt{CMC} | \texttt{arriveQ} : \texttt{amsq} , \texttt{sendQ} : \texttt{smqs} \rangle$ , \\
$\langle q : \texttt{CMgrC} | \texttt{in} : \texttt{imq} , \texttt{out} : \texttt{omq} , \texttt{cpConf} : \texttt{cps} , \texttt{conf} : \langle \rho : \texttt{MCC} \rangle \ldots \rangle , \langle \delta : \texttt{CPC} \rangle \ldots \rangle , \texttt{mconf} \rangle$ \\
$q \triangleleft \texttt{arrive}('\texttt{msg}!',\texttt{snr},\texttt{amap})$
\textbf{Step 2} \textit{RCF}($\texttt{cmgr2cma}$) \\
$\langle q : \texttt{CMgrC} | \texttt{in} : \texttt{imq} , \texttt{out} : \texttt{omq} \circ ('\texttt{msg}!',\texttt{snr},\texttt{amap}), \texttt{cpConf} : \texttt{cps} , \texttt{conf} : \langle \rho : \texttt{MCC} \rangle \ldots \rangle , \langle \delta : \texttt{CPC} \rangle \ldots \rangle , \texttt{mconf} \rangle$ \\
\texttt{mconf} \rangle \\
\hline
$\implies$ \\
$\langle q : \texttt{CMgrC} | \texttt{in} : \texttt{imq} , \texttt{out} : \texttt{omq} , \texttt{cpConf} : \texttt{cps} , \texttt{conf} : \langle \rho : \texttt{MCC} \rangle \ldots \rangle , \langle \delta : \texttt{CPC} \rangle \ldots \rangle , \texttt{mconf} \rangle$ , \\
\texttt{cma} $\triangleleft$ \texttt{arrive}('\texttt{msg}!',\texttt{snr},\texttt{amap})
\end{tabular}
\end{center}

Fig. 9. RCF Interface rules

Message handling: When the communication manager, \texttt{q}, receives the forwarded message, it queues into its \texttt{in} queue for later processing (step 1). Thus, we can integrate message queue priority in a straight forward manner by
queueing a message in a particular place. Some queued messages may not require any processing (i.e., they are raw messages), they must be identified and sent to the cm a for message notification and delivery. Since the message service list (msl) is already included in the message, the communication manager can classify messages as processed or raw message by inspecting its msl. Raw messages are queued into the out queue in order to be returned immediately to the cm a (step 2). In order to unwrap a processed message, the communication manager assigns a message coordinator, ρ, to unwrap the processed message (step 3) and the assigned message coordinator queues the request for later processing (step 4). In order to unwrap the message, the message coordinator extracts the message by inspecting its content (step 5). Figure 10 shows the message handling rules described above.

<table>
<thead>
<tr>
<th>Step</th>
<th>RCF Message Handling Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>RRCm^9t(inq)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>RRCm^9t(tunnel)</code></td>
</tr>
<tr>
<td>3</td>
<td><code>RRCm^9t(cmsg2me)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>RRCm^9t(cmsgmail)</code></td>
</tr>
<tr>
<td>5</td>
<td><code>RRCm^9t(extractmsl)</code></td>
</tr>
</tbody>
</table>

**Fig. 10.** RCF message handling rules

**Message processing:** Once the information have been obtained, the message coordinator proceeds to unwrap the message by coordinating the execution order of the protocols as follows (see figure 11): Message coordinator requests to unwrap message to a specific communication protocol (step 6). Note that the communication protocol, δ, only receives the message and after finish its processing, returns the message to the message coordinator, which selects and repeats the process if needed. That is, message coordinator needs to keep track the stage and protocol currently processing the message.

If the protocol is statefull, it processes the message and notifies the message coordinator to expect future information (step 7). Here we use the spin flag to keep the message coordinator spinning and the statefull flag to keep the
protocol state. Note that we use the event handling to notify message coordinator that it needs to spin, but we do not wait for an acknowledgement. In case the protocol is stateless, it processes the message and notifies the message coordinator when it finishes (step 8).

Once the message has been unwrapped, the protocol returns the message to the message coordinator (step 9). In case the message needs to be processed by another protocol, the message coordinator will repeat the process (step 6) with the corresponding protocol; otherwise, the message has been completely unwrapped.

**Fig. 11.** RCF message processing rules

**Message delivery:** Before the message can be returned to the e\textsubscript{ma}, the message coordinator checks if the message was processed by at least one statefull protocol (see figure 12) in order to determine if it needs to wait (spin) for further information (a tag or acknowledgement) before it can process another message (step 10); otherwise it can be reused by the communication manager (step 11). In both cases, the message is returned to the communication manager, which queues the message into its \textit{out} queue (step 12) Once the unwrapped message is in the communication manager’s \textit{out} queue, it can be forwarded to the \textit{e\textsubscript{ma}} (step 2). Once the message is received by the \textit{e\textsubscript{ma}}, message notification and delivery follows the same rules described for the case when the \textit{rcf} module is inactive (step 1 to step 4).

**B Complete Set of Rules**
Step 10 $RR^{mc}(spinm\rightarrow cmgr)$

$\rho : MCC | mailq : finish(\delta, !msg), prlst, rst : rslst, ip : iplst, ord : ordl, spin : false >$

$\implies$

$\rho : MCC | mailq : finish(\delta, !msg), prlst, rst : rslst, ip : iplst, ord : ordl, spin : true >,\$

$q \# processed(\text{msg}, \text{sndr, amap})$

Step 11 $RR^{mc}(mc\rightarrow cmgr)$

$\rho : MCC | mailq : finish(\delta, !msg), prlst, rst : rslst, ip : iplst, ord : ordl, spin : false >$

$\implies$

$\rho : MCC | mailq : none, pr : none, rst : none, ip : none, ord : none, spin : false >,\$

$q \# processed(\text{msg}, \text{sndr, amap})$

Step 12 $RR^{msr}(outq)$

$\rho : CMgr | in : \text{msg}, \ out : omsg, cpConf : cps, conf :< \rho : MCC | \ldots >, \ < \delta : CPC | \ldots >, \ mConf >$

$\implies$

$\rho : CMgr | in : \text{msg}, \ out : omsg \circ (\text{msg}, \text{sndr, amap}), cpConf : cps, conf :< \rho : MCC | \ldots >, \ < \delta : CPC | \ldots >, \ mConf >$

Fig. 12. RCF message delivery rules
Step 1 \( R^{RRCf}_1 \) (1)
\[
\begin{align*}
&\text{Step 2 } R^{RRCf}_2 \text{(2)} \\
&\text{Step 3 } R^{RRCf}_3 \text{(3a)} \\
&\text{Step 4 } R^{RRCf}_4 \text{(4)} \\
&\text{Step 5 } R^{RRCf}_5 \text{(5)} \\
&\text{Step 6 } R^{RRCf}_6 \text{(6)} \\
&\text{Step 7 } R^{RRCf}_7 \text{(7)} \\
&\text{Step 8 } R^{RRCf}_8 \text{(8)} \\
\end{align*}
\]

Fig. 13. TLS-RCF rules: message 1
Fig. 14. TLS-RCF rules: message 2
Step 15 \( RR_{\text{clear}}(13) \)

< CM gr(A): CM gr[C] in m ag + \( [M_2, A, m \text{ sid}, m s l] \), out: cm ag, L olst:s list, \( \text{cpCom f} : \text{leps}, \text{com f} \): \( < m \in \text{MCC} [.] \ldots >, < \text{cp CPC} [.] \ldots >, \text{m com f} >

\[
\]

Step 16 \( RR_{\text{clear}}(14) \)

< m cl: M CC \( m = \text{iqq}(\text{recv}(M_2, A, m \text{ sid}, m s l)) \), pr: pr lst r slst list, t cpiplat, or dordi, st = ge3, spin t r u e >

\[
\]

Step 17 \( RR_{\text{clear}}(15) \)

< cp(TLS): CPC \( \text{and} d q = m \ldots \text{recq}((\text{de crypt}(M_2, A)), m \text{ sid}, m \text{ cid}) \), role: Init i s tor, st cK y : S K A, c i p h e r s u i te: n o n e, s a K e y i n o t e, d c o m : B, e c o m : n o n e, c o u n t e r : J + 1, a s s i s n o n e, s = ge3, s u t e f ul l: t r u e >

\[
\]

Step 18 \( RR_{\text{clear}}(16) \)

< m cl: M CC \( m = \text{iqq}(M_3, B, m \text{ sid}, \text{sp tm c}) \), pr: pr lst r slst list, t cpiplat, or dordi, st = ge3, spin t r u e >

\[
\]

Step 19 \( RR_{\text{clear}}(17) \)

< CM gr(A): CM gr[C] in m ag + \( [M_3, B, m \text{ sid}, m s l] \), L olst:s list, \( \text{cpCom f} : \text{leps}, \text{com f} \): \( < m \in \text{MCC} [.] \ldots >, < \text{cp CPC} [.] \ldots >, \text{m com f} >

\[
\]

Fig. 15. TLS-RCF rules: message 3
Step 20  $RR^{tlsref}_{20}$ (18)

\[
< \text{cm} \text{gr}(B); CMgr\text{C}[in;m \in \{M3, B, \text{msgid, mas}\}, \text{outs}; \text{cm} \text{g}, L0; \text{st}; \text{slat}'; cpl\text{com} \text{f} ; \text{lesp}, \text{com} \text{f} ; < \text{cm} \text{cMC} \text{C} \Rightarrow , < \text{ep} \text{CPC} \text{C} \Rightarrow , \text{m} \text{con} \text{f} > \\
\]

\[
\Rightarrow \\
< \text{cm} \text{gr}(B); CMgr\text{C}[in;m \in \{M3, B, \text{msgid, mas}\}, \text{outs}; \text{cm} \text{g}, L0; \text{st}; \text{slat}'; cpl\text{com} \text{f} ; \text{lesp}, \text{com} \text{f} ; < \text{cm} \text{cMC} \text{C} \Rightarrow , < \text{ep} \text{CPC} \text{C} \Rightarrow , \text{m} \text{con} \text{f} > ,
\]

\[
m = \text{c}24(\text{rcv}(M3, B, \text{msgid, mas}))
\]

\[
if(m = c24): (\text{getm} \text{cF} \text{rom} \text{M} \text{A}(t = \text{cw} \text{c} \text{M} \text{A}(\{M3, B, \text{msgid, mas}\}))
\]

Step 21  $RR^{tlsref}_{21}$ (19)

\[
< \text{cm} \text{c}2\text{MCC} = \text{c}1\text{q}(\text{rec}(M3, B, \text{msgid, mas})); \text{pr}t; \text{rel}; \text{rstat}; \text{cpiplat}; \text{orddiri}; \text{stge} = 2, \text{apin} \Rightarrow \text{t se} > \\
\]

\[
< \text{cm} \text{c}2\text{MCC} = \text{c}1\text{q}(\text{rec}(M3, B, \text{msgid, mas})); \text{pr}t; \text{rel}; \text{rstat}; \text{cpiplat}; \text{orddiri}; \text{stge} = 2, \text{apin} \Rightarrow \text{t se} > ,
\]

\[
\text{cp}(\text{TLS}) \text{e}((\text{decr} \text{rypt}(M3, B)), \text{msgid} = \text{cid})
\]

\[
if((\text{TLS} = \text{eq} \Rightarrow \text{t se} = \text{m} \Rightarrow \text{al}(\text{rcv}(M3, B, \text{msgid, mas})))) \Rightarrow \text{nd}(\text{mc} \text{id} = \text{getm} \text{cid}(m \text{c}24))
\]

Step 22  $RR^{tlsref}_{22}$ (20)

\[
< \text{ep}(\text{TLS}) \text{e} (\text{CPC}) \text{e} \text{nd} \text{q} \text{=} \text{none}, \text{rec} = ((\text{decr} \text{rypt}(M3, B)), \text{msgid} = \text{cid}); \text{ro}k = \text{resp} \text{on} \text{de} \text{r} \text{e} \text{c} \text{y} = \text{SKB} \text{e} \text{c} \text{ip} \text{e} \text{r} \text{s} \text{u} \text{it} \text{e} = \text{c} \text{ip} \text{e} \text{r} \text{s} \text{n} \text{e}, \text{d} \text{e} \text{com} = \text{A}, \text{ec} \text{om} = \text{n} \text{e}, \text{cou} \text{nt} = \text{K} + 1, \text{s} \text{e} \text{s} \text{s} \text{i} \text{on} \text{e} = \text{s} \text{i} \text{l} \text{e} = \text{ge} = 3, \text{st} \text{at} \text{e} \text{f} \text{u} \text{ll} \text{t} \text{r} \text{u} \text{se} > \\
\]

\[
< \text{ep}(\text{TLS}) \text{e} (\text{CPC}) \text{e} \text{nd} \text{q} \text{=} \text{none}, \text{rec} = ((\text{decr} \text{rypt}(M3, B)), \text{msgid} = \text{cid}); \text{ro}k = \text{resp} \text{on} \text{de} \text{r} \text{e} \text{c} \text{y} = \text{SKB} \text{e} \text{c} \text{ip} \text{e} \text{r} \text{e} \text{s} \text{u} \text{it} \text{e} = \text{c} \text{ip} \text{e} \text{r} \text{s} \text{n} \text{e}, \text{d} \text{e} \text{com} = \text{A}, \text{ec} \text{om} = \text{n} \text{e}, \text{cou} \text{nt} = \text{K} + 2, \text{s} \text{e} \text{s} \text{s} \text{i} \text{on} \text{e} = \text{s} \text{i} \text{l} \text{e} = \text{s} \text{i} \text{e} = \text{ge} = 2, \text{st} \text{at} \text{e} \text{f} \text{u} \text{ll} \text{t} \text{r} \text{u} \text{se} > ,
\]

\[
m = \text{c}24(M3, A)
\]

\[
if(((\text{y} = \text{P} \text{h} \text{r} = \text{SKB}, \text{PKB}) \Rightarrow \text{nd}(\text{K} = \text{n} \text{e} \text{o} \text{c} = (\text{K}, \text{K}))) \Rightarrow \text{nd}(\text{dec} = \text{A}) \Rightarrow \text{nd}(\text{new} \text{=} \text{Sid}(\text{A}, K) = \text{nd}(\text{dec} = \text{A}) \Rightarrow \text{nd}(\text{crypt}(\text{finish} = \text{d}, \text{SKEy})))
\]

Step 23  $RR^{tlsref}_{23}$ (21)

\[
< \text{cm} \text{c}2\text{MCC} = \text{c}1\text{q}(M4, A); \text{pr}t; \text{rel}; \text{rstat}; \text{cpiplat}; \text{orddiri}; \text{stge} = 3, \text{apin} \Rightarrow \text{t se} > \\
\]

\[
< \text{cm} \text{c}2\text{MCC} = \text{c}1\text{q}(M4, A); \text{pr}t; \text{rel}; \text{rstat}; \text{cpiplat}; \text{orddiri}; \text{stge} = 3, \text{apin} \Rightarrow \text{t se} > ,
\]

\[
\text{cm} \text{gr}(B) = (M4, A, \text{mas})
\]

\[
if(M4 = \text{eq} \Rightarrow \text{crypt}(\text{finish} = \text{d}, \text{SKEy}))
\]

Step 24  $RR^{tlsref}_{24}$ (22)

\[
< \text{cm} \text{gr}(B); CMgr\text{C}[in;m \in \{M4, A, \text{mas}\}, L0; \text{at}; \text{slat}'; \text{cplcom} \text{f} ; \text{lesp}, \text{com} \text{f} ; < \text{cm} \text{cMC} \text{C} \Rightarrow , < \text{ep} \text{CPC} \text{C} \Rightarrow , \text{m} \text{con} \text{f} > \\
\]

\[
< \text{cm} \text{gr}(B); CMgr\text{C}[in;m \in \{M4, A, \text{mas}\}, L0; \text{at}; \text{slat}'; \text{cplcom} \text{f} ; \text{lesp}, \text{com} \text{f} ; < \text{cm} \text{cMC} \text{C} \Rightarrow , < \text{ep} \text{CPC} \text{C} \Rightarrow , \text{m} \text{con} \text{f} > ,
\]

\[
\text{cm} \text{gr}(A) = (M4, A, \text{mas})
\]

\[
if((\text{cm} \text{gr}(A) = \text{getcm} \text{id}(A))) \Rightarrow \text{nd}(M4 = \text{crypt}(\text{finish} = \text{d}, \text{SKEy}))
\]

Fig. 16. TLS-RCF rules: message 4
Fig. 17. TLS-RCF rules: message 5
Step 30 $RR^{larcf}_{30}$

< Cm gr(B); Cm gr[C] | in: m ag+ ((M5, B, m agid, m sl)), out: com ag, L06; stat, ecpCon f! sesp; com f; < m c; M CC|...>, < ecp CPC|...>, m com f

Step 31 $RR^{larcf}_{31}$

< m c; M CC| m a ilq: note, pr: prist, rst: rstat, st: sp; get; 4, sp: tr ut>

Step 32 $RR^{larcf}_{32}$


Step 33 $RR^{larcf}_{33}$

< m c; M CC| m a ilq: (m 1, B), pr: prist, rst: rstat, st: sp; get; 5, sp: tr ut>

Step 34 $RR^{larcf}_{34}$

< Cm gr(B); Cm gr[C] | in: m ag+ ((M4, A, m sl)), L06; stat, ecpCon f! sesp; com f; < m c; M CC|...>, < ecp CPC|...>, m com f

Step 35 $RR^{larcf}_{35}$

< M gr(B); M gr[C] | up: none, dwm: rÍm ag+ (m 1), do sm: agid, te sm agid, te sm all; m sl>

< B: A c| tr...

< M gr(B); M gr[C] | up: none, dwm: rÍm ag+ (m 1), do sm: agid, te sm agid, te sm all; m sl>

< B: A c| tr | m = dq; m 1, ...

Fig. 18. TLS-RCF rules