A Semantic Framework for Modeling and Reasoning about Reflective Middleware: The Logger Example

Nalini Venkatasubramanian
University of California, Irvine
nalini@ics.uci.edu

Carolyn L. Talcott
Stanford University
clt@cs.stanford.edu

Abstract

In a distributed reflective framework, issues of correctness and composition can be quite subtle and complex. Interactions within and across reflective levels must be considered, the semantics of shared, distributed resources must be clearly spelled out, and new notions of correctness of the overall system need to be developed that account for the dynamic, distributed, and reflective setting. TLAM is a two-level model of distributed computation based on the actor model of object-based distributed computation that supports dynamic customizability and separation of concerns in designing and reasoning about components of Open Distributed Systems. The TLAM uses reification (base object state as data at the meta object level) and reflection (modification of base object state by meta objects) with support for implicit invocation of meta objects in response to changes of base level state. This provides for debugging, monitoring, and other hooks. In this paper we briefly review the TLAM concepts and summarize the application of the TLAM framework to fairly elaborate middleware services. The main contribution of this paper is a simple example worked out in some detail to illustrate techniques for using the TLAM to model middleware services as reflective / meta-level services and to illustrate our multiple viewpoint methodology for specifying and reasoning about such services.

Keywords: open distributed system, reflection, middleware, meta-architecture, composition, specification, verification.

1 Introduction

Open Distributed Systems (ODS) evolve dynamically and components of ODS interact with an environment that is not under their control. A reflective model of distributed computation supports both separation of concerns (for example functionality and different QoS properties) and dynamic adaptation to changing environment or requirements. In such an ODS a wide range of services and activities must execute concurrently and non-disruptively, and must share resources. In order to avoid resource conflicts,
deadlocks, inconsistencies and incorrect execution semantics, the underlying resource management system (middleware) must ensure that the concurrent system activities compose in a correct manner. In a distributed reflective framework, issues of correctness and composition can be quite subtle and complex. Interactions within and across reflective levels must be considered, the semantics of shared, distributed resources must be clearly spelled out, and new notions of correctness of the overall system need to be developed that account for the dynamic, distributed, and reflective setting.

To gain a better understanding of the semantic issues involved in reflective distributed systems, we developed the Two Level Actor Model - TLAM [41, 37], a two-level model of distributed computation based on the actor model of object-based distributed computation [16, 7, 1]. The actor model is a model of distributed reactive objects that has a built-in notion of encapsulation and interaction and is thus well-suited to represent evolution and co-ordination among interacting components in distributed applications. Traditional passive objects encapsulate state and a set of procedures that manipulate the state; an actor encapsulates a thread of control as well. Each actor potentially executes in parallel with other actors and interacts only by sending and receiving messages.

As the name suggests, in the TLAM, a system is composed of two kinds of actors, base-level actors and meta-level actors, distributed over a network of processing nodes. Base-level actors carry out application-level computation, while meta-level actors are part of the runtime system (middleware) that manages system resources and controls the runtime semantics of the base level. The TLAM abstracts from the choice of a specific programming language or system architecture, providing a framework for reasoning about heterogeneous systems. The TLAM supports dynamic customizability and separation of concerns in designing and reasoning about components of ODS. The TLAM uses reification (base object state as data at the meta object level) and reflection (modification of base object state by meta objects) with support for implicit invocation of meta objects in response to changes of base level state. This provides for debugging, monitoring, and other hooks.

1.1 The TLAM Approach to Specification

We have used the TLAM framework is several substantial case studies. Our general approach to modeling middleware components is to develop a family of specifications from different points of view and at different levels of abstraction. From a high-level point of view we specify the end-to-end service provided by a system in response to a request. From a design point-of-view we specify system-wide properties in terms of global states and their possible evolutions. The idea is to identify invariants and responsiveness requirements that both serve as an implementation guide and ensure that the system provides the desired end-to-end service. From a prototyping point-of-view we specify constraints on the behavior and distribution of a group of actors. In addition we define a map from the service interface (requests and replies) to messages to and from certain of the meta-actors, and specify initialization and non-interference conditions required by the meta-actor system for correct operation. This local behavior point of view can be further refined by specifying protocols and algorithms for the actions of individual actors. In addition to defining specifications from the different
points of view, we must show that

(ss-ee) the system-wide requirements imply that the end-to-end specification is satisfied

(beh-ee) using the interface mapping, a group of meta actors with the specified behavior and distribution provides the specified end-to-end service, under the specified initial and non-interference conditions.

(alg-beh) each behavioral refinement or algorithm meets the corresponding behavior constraints

The relation (beh-ee) is usually established by showing (beh-sw) that system-wide requirements are met and appealing to (ss-ee). This staging and refinement of specifications provides a form of modularity, scalability, and reusability. It reduces the task of implementation to that of implementing individual abstract behaviors. Behavior level specifications can be used to guide or check implementations or even serve as executable prototypes. Three major case studies have been carried out using the TLAM framework: (1) Distributed Garbage Collection (DGC) [39, 36]; (2) composition of migration and reachability services [41]; (3) QoS-based resource management for multimedia servers [40].

1.2 The CompOSE|Q Framework

The CompOSE|Q framework (See Fig 1), currently being developed at the University of California, Irvine [38, 13] implements the basic TLAM metaarchitecture with additional features to support a reflective communication layer for actor interaction. The work addresses implementation issues and performance overheads associated with the realization of the TLAM methodology in a general-purpose middleware framework. To ensure non-interference and manage the complexity of reasoning about multiple middleware components and services, we identify three key core services within CompOSE|Q provided by meta-actors where non-trivial base-meta interactions occur: remote creation, distributed snapshots (cf. [10, 11]), and directory services. Using the core services, we define other meta-level services and specify properties such as functionality, quality-of-service and non-interference requirements in terms of purely meta-level interactions, which are more easily expressed and understood, although still non-trivial. Within CompOSE|Q, we have developed modules that implement resource management services such as remote creation, migration and reachability snapshot from formally verifiable specifications developed using the TLAM; the services implement necessary invariants and constraints to ensure the safe composition of meta-level services. The design, implementation and performance of a node-based runtime layer and a distribution infrastructure that implements the necessary TLAM abstractions (meta-actors, base-actors, core services) along with suitable optimizations to improve performance of TLAM-based resource management services are described in [13].

In order to ensure safe and cost-effective QoS in distributed environments, composable of resource management services such as admission control, resource reservation and scheduling is essential. For instance, system level protocols must not cause
arbitrary delays in the presence of timing based QoS constraints. The CompOSE|Q framework implements QoS-based resource management services for distributed multimedia systems as metalevel services in the reflective architecture. Here, we use services developed earlier within the TLAM framework (e.g., migration and snapshot primitives) to develop a QoS brokerage architecture (See Figure 1) and reason about the interaction of services within the architecture [38, 40]. The QoS broker system implements scheduling and routing of multimedia requests as well as placement of multimedia data using a collection of multimedia resource management meta-actors providing these services. While many of the services contained in the metalevel of the CompOSE|Q architecture have been studied in depth; this paper develops a rather simplistic logging service to illustrate specification and reasoning in the TLAM framework in some detail.

The TLAM framework also provides a semantic basis for the development of tools to support modular design and analysis of middleware services with increased assurance: reliability, robustness, and reusability. We are working on a realization of the TLAM framework using the Maude tool [27] based on rewriting logic [26]. The TLAM in Maude constitutes an executable specification that can be used for prototyping, symbolic simulation, and simple analyses (using reflection in Maude).

1.3 Related Work

Research on computational reflection was initiated by work of Brian Smith [33] (3-Lisp) and Patty Maes [24] (3KRS). A number of reflective actor-based languages have been developed to support separation of concerns and high-level programming abstractions for distributed systems. The ABCL family of languages [44] explores different forms of reflection, including single-actor and group-based reflection. A layered reflection model (the onion skin model) is used as the basis for developing coordination primitives [14, 2], representation of dependability protocols as meta-level programs [3, 35],

Figure 1: Architecture of the CompOSE|Q System. The dotted lines indicate the flow of an incoming request through the middleware modules.
and architectural abstractions [6]. Meta-object protocols [19] provide more restricted forms of reflective capability, consisting of interfaces to a language that give users the ability to incrementally modify the language’s behavior and implementation, as well as the ability to write programs within the language. This approach was generalized to the Aspect Oriented Programming paradigm [20] to facilitate separation of concerns, composition, and re-use in programming.

In other reflective models for distributed object computation, e.g. the Multi-Model Reflective Framework (MMRF) [30] an object is represented by multiple models allowing behavior to be described at different levels of abstraction and from different points of view. In each model the behavior of an object is described by a metaspace that consists of meta objects representing the different models/aspects (e.g. encapsulation, environment) and each meta object sees and acts on one base level object. In the TLAM, each meta actor can examine and modify the behavior of a group of base level actors – namely those located on the same node. While some instances of the TLAM may have the many-to-one organization of the multiple model frameworks, the more flexible relation seems appropriate when considering resource management facilities that involve manipulation of collections of base level actors.

Examples of operating systems built using reflective distributed object models are Apertos [17], Legion [15], and 2K [22]. Adaptability and extensibility is a prime requirement of middleware systems; reflective middleware typically builds on the idea of a meta-object protocol, with a meta-level describing the internal architecture of the middleware and reflection used to inspect and modify internal components. DynamicTao [21] is a reflective CORBA ORB [29] built as an extension of the Tao real-time CORBA ORB[31]. DynamicTao supports on-the-fly reconfiguration while maintaining consistency by reifying both internal structure and dependency relations using objects called configurators, which provide interfaces for inspection and modification. Other work on using reflective ORBs to customize resource management behavior, e.g. scheduling is reported in [32]. In [43] the use of reflective middleware techniques to enhance adaptivity in QoS-enabled component-based applications is discussed and illustrated using the Tao ORB. A reflective architecture for next-generation middleware based on multiple meta-models is described in [8, 9] and a prototype has been developed using the reflective capabilities of Python [5]. In contrast, the CompOSE|Q system described earlier is based on a semantic model, the TLAM, that has been used extensively to reason about the composability of services implemented within the system.

One of the main objectives of the TLAM architecture is to develop formal models, reasoning, and analysis techniques to address the challenges of performance and integrity management associated with adaptable and extensible middleware. An associated goal is to provide a semantic basis for separation of concerns and compositional development of middleware for large scale distributed systems. There are two dimensions of composition: that of the base-level with the meta-level, and composition of meta-level services over arbitrary base-level systems. The work on superposition (see [18]) also addresses issues of separation of concerns and compositionality by developing generic linguistic constructs based on notions of role and binding. An interesting question for future investigation is whether the notion of superposition can be used to develop more systematic composition mechanisms within the TLAM frame-
work.

The two-level TLAM architecture naturally extends to multiple levels, with each level manipulating the level below while being protected from manipulation by lower levels. A purely reflective architecture provides an unbounded number of meta-levels with a single basic mechanism. The formal verification of interaction semantics between the different layers in the reflective hierarchy can be quite complex. In order to make progress, we have begun by focusing on the special case of two levels. The challenge here is to develop easy to use principles for harnessing the power of reflection and avoiding the potential chaos that is possible with its unrestricted use. The objective of this paper is to illustrate the use of the TLAM framework, in particular, and the key elements of our approach to modeling middleware services from multiple points of view. A full mathematical definition of the TLAM framework can be found in [42] along with some detailed case studies. In §2 we briefly review the TLAM concepts. In §3 we work out a simple example of a logger service in some detail to illustrate the modeling and reasoning techniques using the TLAM framework. §4 concludes with some directions for future work.

2 The TLAM Framework

The TLAM semantic framework [41, 37, 42] is based on the actor computation model [16, 7, 1]. In the TLAM, a system is composed of two kinds of actors, base-level (application) actors and meta-level (system, middleware) actors, distributed over a network of processing nodes. Meta-actors communicate with each other via message passing as do base-actors, and meta-actors may also examine and modify the state of the base actors located on the same node. Base-level actors and messages have associated runtime annotations (finite sets of tagged values) that can be set and read by meta-level actors, but are invisible to base-level computations. Actions which result in a change of base-level state are called events. In the TLAM, the base-level state can be changed by application of a base-level step rule, which involves only base-level entities, or by application of a meta-level step rule, which may involve both base- and meta-level entities. In addition to changing the actors local state, application of an step rule may result in creation of new actors, or sending of new messages. The TLAM event handling mechanism allows meta-actors to react to base-level events. The TLAM framework provides a notion of model (system description) and an associated semantics.

2.1 TLAM model

A TLAM model is a structure of the form

\[ TLAM = \langle Net, TLAS, loc \rangle \]

where \( Net \) is the underlying network, with processor nodes and communication links, and \( TLAS \) is a two-level actor system with actors distributed over the network by the function \( loc \). The TLAM function \( loc \) maps an actor name to the node on which the actor resides. This corresponds to giving each node a pool of actor names to use when creating new actors, rather than having a global name generator.
The TLAS part of a TLAM model is given by specifying: a set $Act$ of actor names; a set $Ad$ of actor state descriptions; a set $Val$ of of values that can be communicated in messages and placed in annotations; and step and event handling rules. Additional entities are built from the three given sets: actors have a name and a state description; messages $Msg$ have a target actor (an actor name) and a contents (the value being transmitted); and annotations are finite maps from tags (special values) to values. We let $a$ (and other variables to be introduced later) range over $Act$ and $A$ range over $P_w(Act)$ (finite sets of actor names). An actor with name $a$ and state $ad$ is denoted by $(a : ad)$. We let $m$ range over $Msg$ and we write $a \triangleleft v$ for a message with target $a$ and contents $v$. The given and derived sets are each partitioned into base- and meta-level. We write $X\|b$ to denote the base-level partition of the set $X$ and $X\|m$ to denote its meta-level partition. Base-level actors and messages also have annotations. We write $(a : ad[\alpha])$ for an actor with annotation $\alpha$, for example $(a : ad[count = 2])$ is an actor with a single annotation with tag $count$ and value $2$.

A step rule describes a possible action for an actor in a given state. It specifies the message consumed (if any) by the stepping actor, the change of the stepping actors state, and any messages to be sent or actors to be created. In addition meta-level step rules may specify changes in the state of base-level actors, base-level messages to be sent, and base-level actors to be created.

A meta-actor step rule has the general form:

$$
\begin{align*}
(f) \quad a : ad \cdot \mu & \xrightarrow{C_b^\mu, upd} a : ad' \cdot C_{new} \quad \text{if cond}
\end{align*}
$$

where $a : ad$ is the stepping meta-actor with name $a$ in state $ad$, and $\mu$ is either empty or a message $a \triangleleft v$ to $a$ with content $v$. $ad'$ is the new state of the stepping actor and $C_{new}$ specifies the newly created meta-level actors and messages. $C_b$ describes the state of the base-level actors observed by the stepping meta-actor, $C_b^\mu$ describes the modification to those actors along with any new base-level actors or messages to be created, and $upd$ specifies modifications to base-level annotations. $\text{cond}$ is a predicate that may depend on $a$, $v$, $ad$, and $C_b$. Base-level step rules have the same form as meta-level step rules, omitting observation/modification of the state of other base-level actors, that is, $C_b^\mu$, and $upd$ are omitted. Mathematically, a rule is just a relation on the sorts of entities that appear as components. Often we present these relations using schemata, but the TLAM framework does not specify a particular syntax for rule components such as $ad$, $C_b$, or $\text{cond}$.

A step that delivers or sends base-level messages, changes base-level state, or creates new base-level actors that can be observed at the meta-level using events. Formally, an event is a structure containing the observed base-level state, message delivered if any, and the base-level modification. An event handling rule specifies the response of a meta-level actor to and event. An event rule is similar to a meta-level step rule with no message delivered and an added event predicate that determines which events the rule applies to. Furthermore, the only base-level modifications that can be specified are annotation modifications. In particular, application of an event handling rule does not generate an event.
A meta-actor event-handling rule has the general form:

$$
\begin{align*}
\{ & \} \quad \text{a : ad } \xrightarrow{\text{event}} \text{ upd } \text{ a : ad'} \cdot C_{\text{new}} \quad \text{if cond}
\end{align*}
$$

where \( ad' , C_{\text{new}} , \text{ upd} \) and \( \text{ cond} \) are the same as for step rules. \( \text{ event} \) is the event predicate. We often use patterns to describe event predicates. For example, if \( v \) is a meta-variable ranging over values, and \( a \) is a particular base-level actor name, then \( \text{ deliver}(a < v) \) describes the predicate that is true for events in which a message is delivered to the actor named \( a \).

### 2.2 TLAM Semantics: Configurations, Transitions, Computations

The semantics of a TLAM model is given by a labeled transition relation on system configurations. A TLAM configuration, \( C \), has a set of base-level actors, a set of meta-level actors and a set of undelivered messages—some are traveling along communication links and others are held in node buffers. Formally a configuration is represented by three functions \( C = (ac, nq, le) \) where \( ac \) maps the names of actors in the configuration to their state. \( nq \) maps each node to the sequence of messages in the message buffer of that node, and \( le \) maps each link to the sequence of messages in transit in that link. We will represent configurations by listing the actor name-state pairs, and message buffers of interest. For example if \( C = (ac, nq, le) \) then

\[
(a_0 : ad_0) \cdot (a_1 : ad_1) \cdot [\nu : Q] \cdot [\gamma : Q'] \cdot C
\]

represents a configuration \( C' = (ac', nq', le') \) where \( ac' \) is \( ac \) extended by mapping \( a_0 \) to \( ad_0 \) and \( a_1 \) to \( ad_1 \) (we assume that \( a_0 , a_1 \) are not in the domain of \( ac \)), \( nq' \) is \( nq \) extended by mapping node \( \nu \) to the message sequence \( Q \) and \( le' \) is \( le \) extended by mapping link \( \gamma \) to the message sequence \( Q' \). We recall that \( \text{ loc}(a) \) gives the node on which the actor named \( a \) is located.

We use the following functions to extract information from a configuration \( C \):

- \( \text{ getState}(C, a) \) is the state of actor \( a \) in \( C \), thus \( \text{ getState}(C, a) = s \). This is defined just if \( a : s \) occurs in \( C \).
- \( \text{ setValue}(C, a, t) \) is the value in \( C \) of the annotation with tag \( t \) of actor \( a \).
- \( \text{ setValue}(C, a, t, v) \) sets the value of the annotation with tag \( t \) of actor \( a \), returning the updated configuration. Thus \( \text{ setValue}(\text{ setValue}(C, a, t, v), a, t) = v \).

There are two kinds of transition on configurations: communication and execution. Communication transitions move undelivered messages from node buffers to links and from links to node buffers and are the same in every TLAM. An execution transition consists of a computation step taken by a base- or meta-level actor, by applying an enabled step rule, followed by application of all enabled event handling rules (in some order). All actors observed, modified, or created in an execution transition reside on the node of the stepping actor.

A rule of the form \( \{ \} \) (i.e. a step rule) is enabled in a configuration \( C \) if
the meta-actor \( a \) is present with state \( ad \),

- if the message is non-empty, then the specified message must be available in the message buffer of the node where the actor is located, and
- the condition holds.

A rule of the form \( (\dagger) \) (i.e., an event handling rule) is enabled in a configuration \( C \) following a application of a step rule if

- the meta-actor \( a \) is present with state \( ad \), on the node of the stepping (base-meta-) actor,
- the event predicate \( event \) holds of the step event
- the condition \( cond \) holds.

When an enabled step or event-handling rule is applied to a configuration, the actor state is changed from \( ad \) to \( ad' \), the message delivered, if any, is removed, and the configuration is extended by adding on \( \text{loc}(a) \) (the node where \( a \) is located) the actors and messages specified by \( C_{\text{new}} \), and the configuration is further modified by making the base-level modifications described by \( C_b \), \( C_b' \) (if present) and \( \text{upd} \). The remainder of the configuration is unchanged. The transitions induced by the step and event rules of a TLAS are spelled out in detail in [42]. In the next section examples of such transitions will be given.

A computation path is a possibly infinite sequence of labeled transitions. A computation path \( \pi \) is written as a sequence of transitions of the form

\[ \pi = [C_i \xrightarrow{\pi(i)} C_{i+1} \mid i \in \text{Nat}] \]

The \( i \)th transition, \( \pi(i) \) is \( C_i \xrightarrow{\pi(i)} C_{i+1} \) the source \( \text{source}(\pi(i)) \) of the \( i \)th transition is \( C_i \), also called the \( i \)th stage of \( \pi \). The label of an execution transition specifies the stepping actor and the message delivered, if any. The label of a communication transition specifies the node and link involved and the direction. For example, \( 12m(\gamma, \nu) \) labels a transition that moves a message from the link \( \gamma \) to the node \( \nu \). The semantics of a configuration is the set of fair computation paths starting with that configuration. A computation path is fair if: any communication transition that becomes enabled at some stage eventually occurs; any base- or meta-level step the becomes enabled at some point eventually happens or becomes permanently disabled; and any message in the communication system is either eventually delivered or from some stage on there is no step enabled to deliver it.

**TLAM Systems and Properties**

A TLAM system is a set of configurations closed under the transition relation. Thus if configuration \( C \) is in system \( S \) and \( C \xrightarrow{} C' \), then \( C' \) is also in \( S \). Properties of such a system are specified as predicates on computation paths. A property can be a simple invariant that must hold for all configurations of a path, a requirement that
a configuration satisfying some condition eventually arise, or a requirement involving
the transitions themselves. Properties are checked using the properties of the building
blocks for configurations – message contents and actor state descriptions – and of the
TLAM reaction rules that determine the behavior of actors in the system.

Preservation of base-level computation is a simple example of a property satisfied
by meta-level systems that only manipulate base level annotations, for example for
monitoring. The restriction of a configuration to base-level \( C \mid b \) is obtained by erasing
all meta-level actors and messages, and erasing all base-level annotations. The restric-
tion of a computation to base-level \( \pi \mid b \) is the result of restricting each configuration
to base-level and deleting meta-level transitions. Note that, since meta-level actors can
change the base-level state, the resulting sequence of transitions may not be a compu-
tation.

**Definition 2.1 (Preserves Base-level Computation):** We say that in a system \( S \) base-
level computation is preserved if the following holds:

- if \( \pi \) is a fair computation of \( C \) then \( \pi \mid b \) is a fair computation of \( C \mid b \)
- if \( \pi_b \) is a fair computation of \( C \mid b \) then there is a fair computation \( \pi \) of \( C \) such
  that \( \pi_b = \pi \mid b \)

Preservation of base-level computation is a strong requirement. Useful notions of
preservation of base-level behavior can be defined by observing only certain aspects
of the base-level computation. For example, observing messages exchanged between
certain groups of base-level actors, and hiding others.

### 3 Logger Example

In this section we work out a small example in some detail to illustrate how the TLAM
is used to model middleware services as reflective, meta-level services, and to illustrate
our methodology for specification and proof. The example is an incremental logging
service that can be installed to log messages received by a group of base-level actors in
a distributed system and to report the logged messages incrementally upon request. We
represent a log as a finite set of messages. Such a logging service could be used, for
example, for accounting purposes. By annotating the logged messages with sequence
numbers or timestamps, the logging service could be used for fault tolerance [4].

We begin with a specification of the logging service from an end-to-end poin-
t of view (Definition 3.1). From this viewpoint only messages flowing across ‘system
boundaries’ can be observed. In the logger case we assume we can observe log requests
and replies and the sending of messages to loggable actors. We then specify the log-
ging service from a system-wide view (Definition 3.3). Here we are allowed to observe
delivery of messages to loggable actors as well as log requests and replies, but we do
not see any of the meta actors involved in the service. Theorem 3.4 states that a system
that satisfies the system-wide specification of a logging service provides the service
specified in the end-to-end view. Finally we define a logging behavior (Definition 3.9)
and the initial and non-interference conditions (Definitions 3.11 and 3.12) needed for
a system with logging behavior to provide a logging service. Theorem 3.10 states that
logging behavior preserves base-level computation, and theorem 3.13 states that under the given initial and non-interference conditions a system with logging behavior does indeed provide logging service. We sketch the proof of this theorem to illustrate how features of the actor model effect reasoning about possible computations, in particular, the use of the fairness assumption, and how to account for information contained in messages in transit as well as information contained in the state of individual actors.

Incremental logs are requested from the logging service using requests of the form \( \text{GetLog}(c) \), and replies of the form \( \text{LogReply}(c, \text{log}) \), where \( c \) is the address where the incremental log should be sent, and \( \text{log} \) is the increment reported. These constitute the logging service interface.

**Definition 3.1 (Logging Service: End-to-end View):** A system \( S \) provides incremental logging service with respect to request function \( \text{GetLog} : \text{Act} \rightarrow \text{Msg} \), reply function \( \text{LogReply} : \text{Act} \times P_w(\text{Msg}) \rightarrow \text{Msg} \), and base-level loggable actors \( A_b \), if the following hold:

1. (ee1) For each request \( \text{GetLog}(c) \) that is made of the system there is a uniquely associated reply, \( \text{LogReply}(c, \text{log}) \), emitted by the system.

2. (ee2) The only messages of the form \( \text{LogReply}(c, \text{log}) \) emitted by the system are those associated to a request of the form \( \text{GetLog}(c) \) as in (ee1).

3. (ee3) If a reply \( \text{LogReply}(c, \text{log}) \) is emitted by the system then:
   (a) all messages in \( \text{log} \) are messages that have been sent to actors in \( A_b \)
   (b) If \( \text{LogReply}(c, \text{log}) \) and \( \text{LogReply}(c', \text{log}') \) are distinct replies emitted by the system, then \( \text{log} \cap \text{log}' = \emptyset \).

4. (ee4) if always there is eventually a new log request sent to the system, then every message sent to an actor in \( A_b \) will occur in some log reply.

5. (ee5) the logging service preserves base-level behavior.

Note that (1) and (2) say that there is a bijection between log requests and log replies (preserving the requestor name), that is, there will be exactly one reply to each request. Furthermore each message sent to a loggable actor is reported at most once, and if requests for logs continue to arrive, then each such message will eventually be reported. We have given an informal definition of the end-to-end service. It can be quite naturally represented in a branching time temporal logic such as that developed in [12].

To state the system-wide view of our logging service we define two functions on computation paths to keep track of the accumulated deliveries and log reports. \( \Delta(\pi, i) \) is the set of messages delivered to an actor in \( A_b \) in transitions \( \pi(j) \) for \( j < i \). \( \Lambda(\pi, i) \) is the union of the set of messages sent to clients in log request replies in transitions \( \pi(j) \) for \( j < i \). These functions are defined in terms of their single-step restrictions \( D \) and \( L \).
Definition 3.2 (Logging Functions):

\[
D(\pi, i) = \begin{cases} 
  \{m\} & \text{if } \pi(i) = C_i \xrightarrow{\text{deliver}(a \cdot \mathbf{v})} C_{i+1} \quad \text{where } a \in A_b \\
  \emptyset & \text{otherwise}
\end{cases}
\]

\[
\Delta(\pi, i) = \bigcup_{j < i} D(\pi, j)
\]

\[
L(\pi, i) = \begin{cases} 
  \log & \text{if } \pi(i) = C_i \xrightarrow{\text{send}(\text{LogReply}(c \cdot \mathbf{log}))} C_{i+1} \\
  \emptyset & \text{otherwise}
\end{cases}
\]

\[
\Lambda(\pi, i) = \bigcup_{j < i} L(\pi, j)
\]

Definition 3.3 (Logging Service: System-Wide View): A system \( S \) satisfies the system-wide logging service requirements with respect to request function \( \text{GetLog} : \text{Act} \times \text{Msg} \rightarrow \text{Msg} \), reply function \( \text{LogReply} : \text{Act} \times \text{P}_w(\text{Msg}) \rightarrow \text{Msg} \), and base-level loggable actors \( A_b \), if for each computation \( \pi = [C_i \xrightarrow{i} C_{i+1} \mid i \in \text{Nat}] \) of \( S \) the following hold:

(sw1) There is a bijection between the log request and log reply messages occurring in \( \pi \).

(sw2) \( \Lambda(\pi, i) \subseteq \Delta(\pi, i) \) for \( i \in \text{Nat} \).

(sw3) \( L(\pi, i) \cap L(\pi, j) = \emptyset \) if \( i \neq j \).

(sw4) for each \( i \in \text{Nat} \) and each \( m \in \Delta(\pi, i) \) either there are that no pending log requests after stage \( i \) in \( \pi \) or there is some \( j > i \in \text{Nat} \) such that \( m \in \Lambda(\pi, i) \).

(sw5) The logging service preserves base-level behavior.

Theorem 3.4 (SystemWide implies End-to-end): If system \( S \) satisfies the system-wide requirements for a logging service with respect to request function \( \text{GetLog} : \text{Act} \times \text{Msg} \rightarrow \text{Msg} \), reply function \( \text{LogReply} : \text{Act} \times \text{P}_w(\text{Msg}) \rightarrow \text{Msg} \), and base-level loggable actors \( A_b \), then it provides incremental logging service with respect to these parameters.

Proof: (ee1-2) follows from (sw1), (ee3) follows from (sw2-3), and (ee4-5) follow from (sw4-5) respectively.

To specify logging behavior we specify meta-actor behaviors and configure a service based on these behaviors. We use three kinds of meta actors:

- a **logger** on each node, that does the recording for the loggable actors on its node,
- a **reporter** on each node, that does the reporting for the loggable actors on its node, and
- a **log server** (one for the system) that receives log requests, collects reports from each node in the system and sends the log reply.
The logger, reporter, and log server behaviors are each specified by defining a set of states and rules governing their response to messages and events. We define the states by giving constructors, taking the states to be (freely) generated by applying the constructors to appropriate arguments. We begin by defining the data types for the contents of messages used by meta-actors providing the logger service.

**Definition 3.5 (Logger Messages):** There are request and reply constructors for communication between the log server and clients:

- \( \text{getLog} \circ \_ : \text{Act} \rightarrow \text{Val} \)
- \( \text{getLogReply} : \text{P}(\text{Msg} \rightarrow \text{Val}) \)

Thus a log request message from a client \( c \) to a server \( s \) has the form \( s \circ \text{getLog} \circ c \). There are also request and reply constructors for communication between the log server and the reporters on each node.

- \( \text{sendLog} : \text{Act} \rightarrow \text{Val} \)
- \( \text{report} \circ \_ \circ \_ : \text{P}(\text{Msg} \rightarrow \text{Act} \rightarrow \text{Val}) \)

The meta-level actor name in a \( \text{report} \) message is expected to be that of the sender. It is included in order that a log server managing several reporters will know which reporter each reply comes from.

**Definition 3.6 (Logger Behavior):** There is one constructor of Logger states:

\[ \text{Logger} : \text{P}(\text{Act} \rightarrow \text{Ad}) \]

and a logger meta-actor for loggable actors \( A_b \), with name \( a \) has the form

\[ (a : \text{Logger}(A_b)) \]

The only logger rule is a rule for handling events that deliver a message to one of its loggable actors.

\[ \frac{\text{deliver}(a \circ b)}{\text{setA}(a, \text{Log} \cup b) \rightarrow (a : \text{Logger}(A_b)) \quad \text{if } a \in A_b} \]

This rule says that if \( a \circ b \) has been delivered to some \( a \in A_b \) then the Log annotation of \( a \) will be updated by adding \( a \circ b \) to its value (\( \text{setA}(a, ...) \)). The state of \( a \) is unchanged.

**Definition 3.7 (Reporter Behavior):** There is one constructor for reporter states:

\[ \text{Reporter} : \text{P}(\text{Act} \rightarrow \text{Ad}) \]

and, a reporter meta-actor for loggable actors \( A_b \) with identifier \( a_r \) has the form

\[ (a_r : \text{Reporter}(A_b)) \]

The only reporter rule is a step rule for handling requests (from the log server) to report its current incremental log.

\[ \frac{\text{sendLog}(c)}{\text{setA}(a, \text{Log} \cup b) \rightarrow (a_r : \text{Reporter}(A_b))} \quad \text{if } a \in A_b \]
\[(a_r: \text{Reporter}(A_b)) \cdot c \leftarrow \text{report(log)} @ a_r \quad \text{where} \quad \log = \bigcup_{a \in A_b} \text{getA}(a, \text{Log}).\]

This rule says that when a sendLog message is delivered to \(a_r\), then \(a_r\) will reply with a report message containing the union of the Log attributes for actors in \(A_b\) along with its name as the reply sender. The Log attributes of each actor in \(A_b\) is reset to \(\emptyset\) \((\text{setA}(a, \text{Log}, \emptyset) \mid a \in A_b)\) and the state of \(a_r\) is unchanged.

**Example Logger Computation**

Before defining the log server behavior, let us look at a fragment of a simple computation. This will illustrate both the structure of configurations and how rules describe transitions between configurations. We start the example computation in a configuration \(C_0\) that contains, on node \(\nu\), a logger \(a_l\), a reporter \(a_r\), two loggable base-actors \(a_0\) and \(a_1\), a message for \(a_0\), and other unspecified actors and messages possibly on other nodes (denoted by \(C_x\)).

\[C_0 = (a_0 : ad_0[\text{Log} = Q]) \cdot (a_1 : ad_1[\text{Log} = \emptyset]) \cdot [\nu : a_0 \leftarrow \nu_0] \cdot C_z\]

where \(C_z = (a_l : \text{Logger}([a_0, a_1])) \cdot (a_r : \text{Reporter}([a_0, a_1])) \cdot C_x\)

Note that the state of \(a_0\) in \(C_0\) is \(\text{ad}_0\) and that \(a_0\) has a single annotation with tag Log and value \(Q\). To simplify matters, we assume that the logger and reporter are the only meta-actors on node \(\nu\). In transition 0 a message containing \(\nu_0\) is delivered to \(a_0\). The new state \(\text{ad}_0'\) of \(a_0\) and the message sent \(a_2 \leftarrow \nu_2\) are computed from \(\text{ad}_0\) and \(\nu_0\) using a base-level rule not specified here.

\[(0) \quad C_0 \xrightarrow{\text{deliver}(a_0 \leftarrow \nu_0)} C_1\]

where \(C_1 = (a_0 : \\text{ad}_0'[\text{Log} = Q \cup \{a_0 \leftarrow \nu_0\}]) \cdot (a_1 : \text{ad}_1[\text{Log} = \emptyset]) \cdot [\nu : a_2 \leftarrow \nu_2] \cdot C_z\)

The event associated to the base-level transition satisfies the event predicate deliver\((a \leftarrow \nu)\) of the logger event handling rule for \(a \in \{a_0, a_1\}\). Applying this rule, the delivery is logged using the specified annotation update \(\text{setA}(a_0, \text{getA}(a_0, \text{Log}) \cup \{a_0 \leftarrow \nu_0\})\).

As a result, the Log annotation of \(a_0\) in \(C_1\) is \(Q \cup \{a_0 \leftarrow \nu_0\}\). In (multi) transition 1 the message to \(a_2\) moves into the network, and a sendLog request is generated and moves through the network to the head of the link \(\gamma\) whose target node is \(\nu\).

\[(1) \quad C_1 \xrightarrow{\downarrow} C_1'\quad [\nu : \emptyset] [a_\gamma \leftarrow \text{sendLog} @ \text{logS}] \xrightarrow{\downarrow 2n(\gamma, \nu)} C_2\]

where \(C_2 = (a_0 : \text{ad}'_0[\text{Log} = Q \cup \{a_0 \leftarrow \nu_0\}]) \cdot (a_1 : \text{ad}_1[\text{Log} = \emptyset]) \cdot [\nu : a_r \leftarrow \text{sendLog} @ \text{logS}] \cdot C_z'\)

A transition labeled \(\downarrow 2n(\gamma, \nu)\) moves the message from link \(\gamma\) into the message buffer of node \(\nu\). \(C_z'\) represents changes elsewhere in the system due to the sending of the sendLog request (and possibly other activity). In transition 2 the sendLog request is delivered to \(a_r\) and a reply is sent according to the step rule for the reporter.

\[(2) \quad C_2 \xrightarrow{\text{send}(a_r)} C_3\]

where
The annotation of \( a_0 \) is set to \( \emptyset \) using the update part of the reporter step rule and 
\[
log = getA(C_3, a_0, Log) \cup getA(C_3, a_1, Log)
\]
is the union of the Log annotation of the loggable actors.

Now we define the states and rules for the log server meta-actor, and say what it means for a system to have Logging Service Behavior.

**Definition 3.8 (LogServer Behavior):** A log server meta-actor is either idle and ready to accept log requests, or it is processing a request and waiting for replies from some of its reporters. In both cases it knows the set of reporters that it manages, and in the waiting case it remembers the requestor, the log reports accumulated so far, and the set of reporters whose reports are missing. Thus we define two state constructors for log server meta-actors:

- \( LogServer I : P_o(Act[m]) \rightarrow Ad[m] \)
- \( LogServer W : P_o(Act[m] \times Act[m] \times P_o(Msg[b]) \times P_o(Act[m]) \rightarrow Ad[m] \)

\( (ls : LogServer I(R)) \) is a log server meta-actor ready to handle a request and \( R \) is the set of log reporter actors that it manages. \( (ls : LogServer W(R, c, log, missing)) \) is a log server meta-actor waiting for reports, \( R \) is the set of log reporter actors that it manages, \( c \) is the log requestor, \( log \) is the union of the log reports received so far, and \( missing \) is the set of reporters for which reports have not yet been received.

There are three Log Server rules:

\[
(ls : LogServer I(R)) \cdot ls \leftarrow \text{getLog} \circ \text{client} \rightarrow
\]
\[
(ls : LogServer W(R, \text{client}, \emptyset, R)) \cdot \{ r \leftarrow \text{sendReport} \circ ls \mid r \in R \}
\]

\[
(ls : LogServer W(R, \text{client}, log, R' \cup \{ r \})) \cdot ls \leftarrow \text{report}(log') \circ r \rightarrow
\]
\[
(ls : LogServer W(R, \text{client}, log \cup log', R')) \quad \text{if } r \notin R'
\]

\[
(ls : LogServer I(R)) \cdot \text{client} \leftarrow \text{getLogReply}(log) \circ ls
\]

The first rule says that when a log server is idle it may receive a log request. The request is handled by sending \( \text{sendLog} \) messages to each of its managed reporters and moving to a state in which it is waiting for a reply from each reporter, with the accumulated log initially empty. The second rule says that when a log server is waiting for reports it may receive a missing report (\( r \) is in the missing set). It adds the reported log (\( log' \)) to its accumulated log (\( log \)) and removes the reporter’s name from the missing set. \( r \notin R' \) is needed to ensure that \( r \) is really removed. The third rule says that a log server waiting with missing set empty may send the accumulated log to the client and become ready for the next request.
Definition 3.9 (Logging Behavior Specification): A system $S$ has logging behavior with respect to a log server meta actor $logS$, a set of loggable actors $A_b$, and logging meta actors $LogMA(\nu)$ and reporting meta actors $ReportMA(\nu)$ for $\nu \in Node$, if for $C$ in $S$ and $\nu \in Node$:

- the state of $LogMA(\nu)$ in $C$ ($\text{getState}(C, LogMA(\nu))$) is $Logger(A_b[\nu])$.
- the state of $ReportMA(\nu)$ in $C$ ($\text{getState}(C, ReportMA(\nu))$) is $Reporter(A_b[\nu])$.
- the state of $logS$ in $C$ ($\text{getState}(C, logS)$) is either $\text{LogServerI}(R)$ or $\text{LogServerW}(R, c, log, R')$ where $R = \{ReportMA(\nu) \mid \nu \in Node\}$, $R' \subseteq R$, $c \in Act_m$, and $log \in P_w(Msg[b])$ such that messages in $log$ are addressed to actors in $A_b$.

where $A_b[\nu] = \{a \in A_b \mid loc(a) = \nu\}$.

In the following we fix logging behavior parameters, $A_b$, $logS$, and $LogMA(\nu)$, $ReportMA(\nu)$ for $\nu \in Node$, as in definition 3.9 and let $LMA$ denote the set of log service meta actors

$$LMA = \{logS\} \cup LogMA(Node) \cup ReportMA(Node)$$

A non-logging meta-actor is thus any meta actor not in $LMA$.

Theorem 3.10 (Logging Preserves Base-Level Behavior): If system $S$ has Logging Behavior, then configurations in which the only meta actors are logging meta-actors of $S$ preserve base-level behavior.

Proof: This is easy to see, because only base-level annotations can be effected by steps of meta-actors in $LMA$, and fairness means that the base-level can not be prevented from progressing by infinitely many log requests.

To state the result that logging behavior provides logging service we need two additional definitions, the initial conditions for logging (Definition 3.11) and the non-interference requirements for logging (Definition 3.12).

Definition 3.11 (Logging Initial Conditions): A configuration $C$ satisfies the logging initial conditions if the value of the $Log$ annotation is the empty set for each loggable actor ($a \in A_b$), and the only undelivered messages to logging meta-actors are logging requests addressed to the log server $logS$.

A system $S$ satisfies the logging initial conditions (relative to the logging behavior parameters) if each configuration is reachable by TLAM transitions for the system from a configuration that satisfies the logging initial conditions.

Definition 3.12 (Logging Non-Interference Requirement): A system $S$ satisfies the Logging Non-Interference Requirement if:

- non-logging meta actors do not set $Log$ annotations
- the only messages sent to logging meta actors by non-logging meta actors are log request messages addressed to the log server $logS$. 

16
The non-interference conditions are typical of the minimal non-interference conditions for a meta-level service: the annotations used by a service are controlled by that service, and there is no spoofing of communications internal to the meta-actors providing the service.

Now we can state the main result about Logging Behavior.

**Theorem 3.13 (Logging Behavior Provides Logging Service):** If system $S$ has Logging Behavior, relative to the logging behavior parameters, and satisfies the Logging Initial conditions and the Logging Non-interference requirements, then $S$ provides Logging Service for $A_n$ with request

$$GetLog(\text{client}) = logS \diamond getLog \circ \text{client}$$

and reply

$$LogReply(\text{client, log}) = \text{client} \diamond getLogReply(\text{log}) \circ logS$$

**Proof:** By Theorem 3.4 it suffices to show that $S$ satisfies the system-wide logging service requirements. This follows from lemmas 3.15, 3.16, and 3.17 (given below). Lemma 3.15 establishes an invariant on configurations, $LogOk(C)$, that expresses the key properties of the overall state of a system satisfying the logging initial conditions by accounting for messages in transit as well as local actor states. Lemma 3.16 establishes responsiveness to requests by showing that the reporting process terminates, leaving the log server ready for another request. This combined with the fact that messages are delivered at most once gives the desired bijection property. Lemma 3.17 establishes the accuracy of the service—that each delivered loggable message is reported at most once, and exactly once if requests continue to arrive. □

Logger system internal messages are the messages used for communication between logger meta-actors, that is, messages of the form: $ReportMA(\nu) \diamond sendLog \circ logS$ or $logS \diamond report(\text{log}) \circ ReportMA(\nu)$.

**Definition 3.14 (LogOk):** The invariant $LogOk(C)$ holds just if conditions (1) and (2) below hold.

1. If the log server is idle, with state $LogServeI(R)$ in $C$, then there are no undelivered internal logger system messages.

2. If the log server is waiting for reports, with state $LogServeW(R, c, log, missing)$ in $C$, then there are no undelivered messages to or from the reporters in $R - missing$, and for each reporter $ReporterMA(\nu)$ in $missing$ exactly one of the following holds:

   a. there is a single undelivered message of the form $ReportMA(\nu) \diamond sendLog \circ logS$, or
   b. there is a single undelivered message of the form $logS \diamond report(\text{log}) \circ ReportMA(\nu)$

**Lemma 3.15 (LogOk invariance):** If $S$ satisfies the hypotheses of Theorem 3.13, then for any configuration $C$ of $S$, $LogOk(C)$ holds.
Proof: By the Logging Initial Condition requirement, every configuration in $S$ is reachable in a finite number of transitions from some configuration satisfying the initial conditions. Thus we may reason by induction on the number of transitions. If $C$ satisfies the logger initial conditions, then we are in case (1) of the definition of $LogOk$ and initiality guarantees the absence of internal logger messages. Now suppose that $LogOk(C)$ holds and $\tau = C \xrightarrow{t} C'$. We need only consider execution transitions with stepping actor in LMA since, using the non-interference requirement, other transitions do not effect the $LogOk$ property. If $\tau$ delivers a log request to $logS$, then it must be the case that $logS$ is idle in $C$ and waiting in $C'$. Thus the only internal logger messages in $C'$ are the sendReport messages sent in the transition, one to each reporter, and hence $LogOk(C')$ holds. If $\tau$ delivers a sendReport message, then this message is removed and a report message from that reported is added to the undelivered messages, and since no logger meta-actor state changes $LogOk$ is preserved. If $\tau$ delivers a report then the reporter name is removed from missing and there are no remaining messages to or from that reporter, and again $LogOk$ is preserved. Finally if $\tau$ sends a log reply to a client this must be in a situation where there are no undelivered internal logger messages.

Lemma 3.16 (Termination): If $S$ satisfies the hypotheses of Theorem 3.13, $C$ in $S$, and $\pi$ a fair path for $C$, then if $\pi(i)$ delivers a log request $\logS \triangleleft \text{getLog} c$, then there is some $j > i$ such that in $\pi(j)$ a reply $c \triangleleft \text{getLogReply}(log) \triangleright \logS$ is sent.

Proof: This follows from lemma 3.15, the fairness of the message delivery system and the fact that each reporter sends a reply in the same transition that a sendLog request is delivered.

Logging accuracy is a consequence of the relation between $\Delta$ and $L$ established in the following lemma.

Lemma 3.17 (Logging accuracy): If $S$ satisfies the hypotheses of Theorem 3.13, and $\pi = [C_i \xrightarrow{t} C_{i+1} \mid i \in \text{Nat}]$ is a fair path in $S$ (starting from an initial configuration) we have that

(1) $log \subseteq \Delta(\pi, i)$ if one of the following holds in $C_i$:
   - $log$ is Log annotation of $a \in A$, or
   - $\logS \triangleleft \text{report}(log) @ \text{ReporterMA}(\nu) \triangleright c \triangleleft \text{getLogReply}(log) \triangleright \logS$ is among the undelivered messages, or
   - the state of $\logS$ is $\text{logServeW}(R, c, log, R')$

(2) For any $m \in \Delta(\pi, i)$ exactly one of the following holds:
   - (2.1) $m$ is in the Log annotation of its target actor
   - (2.2) there is an undelivered internal logger report $\logS \triangleleft \text{report}(log) @ \text{ReporterMA}(\nu)$ such that $m \in log$ (and the target of $m$ is located on $\nu$)
(2.3) the state of log $S$ in $C_i$ is $logServeW(R,e,log,R')$ with $m \in log$
(2.4) There is a unique $j < i \in L(\pi,j)$

(3) Furthermore, if case 2.2 or 2.3 holds, or if a log request arrives after stage $i$, then at some stage $j > i$ case 2.4 will hold for $m$

**Proof:** The proof is by induction on $i$. The case $i = 0$ is trivial, $\Delta$ and the Log annotations empty, the server is idle, and there are no outstanding report messages. Assume that the conditions hold for $i$. If $\pi(i)$ delivers a $m$ to $a \in A_1$, then $\Delta(\pi,i+1) = \Delta(\pi,i) \cup \{m\}$ and condition (1) holds for $m$ by the logger rule. sendReport delivery transitions move messages from condition (2.1) to condition (2.2) report delivery transitions move messages from condition (2.2) to condition (2.3), and transitions that send a reply to a client move messages from condition (2.3) to condition (2.4). Each of the transitions also preserves (1) and other transitions do not affect (1) or (2).

Note that (1) implies $\Delta(\pi,i) \subseteq \Delta(\pi,i)$ and (2.4) implies that $L(\pi,i) \cap L(\pi,j) = \emptyset$ if $i \neq j$, i.e., messages are reported at most once.

## 4 Future Research Directions

In this paper, we have illustrated the TLAM approach to specifying and reasoning about reflective middleware services. The TLAM framework is based on a two-level model of open distributed computation and has been used to specify, compose, and reason about a variety of distributed resource management services and their interaction.

Currently, TLAM reflection (modification of base object state by meta objects) provides support for implicit invocation of meta objects in response to changes of base level state. An important topic for future work is the extension of the model to support signals from the base to the metalevel to allow for the explicit invocation of metalevel services by base objects.

Another direction of future work is to extend the TLAM to provide for specification of a wider range of meta-level services, including scheduling, routing, name services, failure semantics, and security. For example, by modeling the scheduler explicitly and providing access to it we can introduce the notion of priorities, preemption, and real-time into the two-level system.

Using generalized state capture facilities within the TLAM framework, we are developing support for fault tolerant systems, e.g., a checkpointing service for capturing causal orders of executions in the system that can be used for monitoring and debugging distributed computations. We are also currently exploring the use of a directory core service to develop policies for resource discovery, group-based communication, access control and security.

The TLAM framework has a very natural representation in rewriting logic [23, 28]. A one-level framework (called actor theories), restricted to purely base-level systems has been developed and applied to specification and reasoning about actor systems and languages [25, 34]. Developing such formal notations or logics is of great interest as a topic of future work.
References


