# Security and Fault-Tolerance in Distributed Systems: An Actor-Based Approach

Gul A. Agha and Reza Ziaei
Open Systems Laboratory
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave,
Urbana, IL 61801 – USA
`{agha|ziaei}@cs.uiuc.edu`
http://www-osl.cs.uiuc.edu/

### Abstract

*The inherent complexity of real-world distributed applications makes developing and maintaining software for these systems difficult and error-prone. We describe an actor-based meta-level model to address the complexity of distributed applications. Specifically, meta-level framework allow code implementing different design concerns to be factored into separate modules – thus enabling the separate development and modification of code for different non-functional requirements such as security and fault-tolerance. This paper reviews current research based on the model and outlines some research directions.*

## 1 Introduction

Real-world distributed applications consist of many asynchronously operating components communicating over networks. They are open to interaction with their environment. Besides satisfying sophisticated functional requirements, such systems are subject to rigorous non-functional requirements such as coordination, timing, fault-tolerance and security. The large number of requirements and their evolutionary nature adds an extra dimension to the complexity of real-world system design. For instance, an avionic control system can be seen as an open distributed system involving a large number of controlling components that interact asynchronously. The specification of such systems usually consists of thousands of requirements that may be modified dynamically to adapt to changes in the environment, control policies, or mission plans.

We believe the problem of maintaining software can be simplified by enabling a modular implementation of distinct design concerns, i.e., the code implementing the functionality of an application should be separated from the code implementing non-functional requirements. A large number of mechanisms have been devised to meet different non-functional requirements, and different mechanisms are suited for different classes of applications. Since the non-functional requirements that we are interested in capturing are ones that affect to the interaction of distributed components, we call such mechanisms *interaction protocols*.

A composable, reusable, and transparent implementation of interaction protocols greatly simplifies software development and maintenance. Because traditional architectures do not allow separate customizable modules for synchronization, scheduling, back-up and error recovery, etc., it is impossible to develop such modular implementations for non-functional requirements.

The goal of this paper is to review the state of the art in Actor research, and to outline research directions. Individual results reported here have been reported in previous publications by the Illinois Open Systems Lab (e.g., [3, 5, 20, 18]). Rather than provide complete technical details here, we refer the reader to these publications.

The outline of this paper is as follows. The next describes the Actor model of concurrency. The model provides a representation for computation in open distributed systems and methods for reasoning about applications in such systems. We then introduce a meta-level framework which allows interaction protocols and application software to be implemented separately and composed. Finally, we discuss how security and dependability protocols may be specified and analyzed using this model.

## 2 Actors

Actors [3] provide a general and flexible model of concurrent and distributed computation. As atomic units of computation, actors may be used to build typical architectural elements including procedural, functional, and object-oriented components. Moreover, actor interactions may be used to model standard distributed coordination mechanisms such as remote procedure call (RPC), transactions, and other forms of synchronization [4, 21, 9, 23]. Modern languages are readily extended with the actor primitives (*cf.* [13]).

Conceptually, each actor has a unique name and an associated behavior. The behavior encapsulates the actor's state and procedures to manipulate the state. Actors communicate by sending messages to one another. Actors compute by serially processing messages sent to them. The serial processing of messages represents a thread of control for the actor. When an actor is idle, it waits for the next message sent to it (see Figure 1).

While processing a message, an actor may perform three types of basic actions that potentially affect computation; namely:

- *send* messages asynchronously to other actors;
- *create* actors with specified behaviors; and
- become *ready* to receive the next message.

Communication between actors is point-to-point and assumed to be weakly fair: executing a *send* eventually causes the message to be delivered to the recipient, provided the recipient is not permanently blocked (e.g., doing an internal computation that results in an infinite loop). The sender need not be blocked waiting for messages that have been sent to

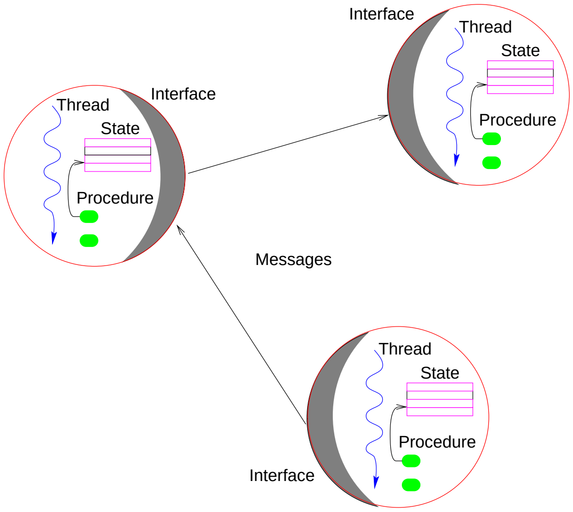


**Figure 1. Actors encapsulate state and procedures to manipulate state. Each actor has a thread of control and multiple actors can run concurrently.**

be delivered. Different messages arrive in an arbitrary order. The *create* primitive creates a new actor with a specified behavior. Initially, only the creating actor knows the name of a newly created actor. Actor names are communicable values, thus other actors can find out the names of newly created actors and the configuration of actors' interconnections may change dynamically. The *ready* primitive is used to indicate that the signaling actor is ready to process the next message in its mail queue. Upon invoking *ready*, the calling actor either begins processing the next available message, or waits until a new message arrives in its mail queue.

Actors are a formal model for concurrent objects: an actor is an object with a private local state, a set of *methods*, a unique name, and a thread of control. Message passing in actors can be viewed as the asynchronous invocation of methods, and standard method invocation can be seen as passing continuations in a message. Serialization of message delivery by an actor provides an object with the semantics of a monitor (without condition variables). Note that the operations *send* and *create* are explicit requests, while *ready* is implicit – method completion frees the monitor and permits processing of the next message.

## An Actor Language and its Semantics

One can extend a sequential language with actor constructs. For example, the call-by-value $\lambda$-calculus is extended in [2]. Specifically, extension adds the following operations to the language:

$\mathtt{send}(a, v)$ creates a new message with content $v$ whose receiver is the actor with name $a$.

`newactor`$(b)$ creates a new actor with initial behavior $b$ and returns its address.

`ready`$(b)$ captures local state change: the behavior of the actor executing the `ready` expression becomes $b$ and the actor is then free to accept another message.

Instantaneous snapshots of actor systems are called *configurations*. An actor configuration is a collection of actors together with undelivered messages. Actor computation is defined by a transition relation on configurations. The notion of open systems is captured by allowing configurations to be composed into new configurations. Composition is meaningful only when configurations have matching interfaces. An interface comprises a set of *receptionists* that can receive messages from actors outside a configuration, and a set of actors *external* to a configuration that can receive messages from the actors within. Interfaces are dynamic and they change as messages pass into or out of a configuration. Formally, an actor configuration is defined as follows:

**Definition (Actor Configurations):** An *actor configuration* with actor map, $\alpha$, multiset of messages, $M$, receptionists, $\rho$, and external actors, $\chi$, is written

$$\langle \alpha \mid M \rangle^{\rho}_{\chi}$$

where $\rho, \chi$ are finite sets of actor names, $\alpha$ maps a finite set of actor names to the corresponding actor behavior, $M$ is a finite multiset of undelivered messages, and if $A = \mathrm{Dom}(\alpha)$, that is $A$ is the domain of actor names, then:

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$,

(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, where $\mathrm{FV}(\alpha(a))$ represents the free variables in $\alpha(a)$, the behavior of actor $a$, and if $\langle a \Leftarrow v_1 \rangle$ is a message with content $v_1$ to actor name $a$, then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

For an actor with name $a$, we indicate its states as $[e]_a$.

Now we can extend the local transitions defined for a sequential language ($\overset{\lambda}{\mapsto}$), by providing transitions for the actor program as shown in Figure 2. Assume that $R$ is the reduction context in which the expression currently being evaluated occurs. Note that in the call-by-value $\lambda$-calculus expressions can be uniquely decomposed into a reduction context and a reduction expression (redex). Thus the behavior of an actor in response to a message is deterministic. The nondeterminism in actor systems results from the indeterminacy in the arrival order of messages.

Based on a variation of the transition system described above, a rigorous theory of actor systems is developed in [2]. Specifically, in that work several notions of equivalence over actor expressions and configurations are defined. The model assumes fairness, namely that any message will be delivered unless its recipient fails. Fairness is an important requirement for reasoning about eventuality properties. It is particularly relevant in supporting modular reasoning. One important consequence of fairness is that each actor makes progress independent of how busy the other actors are. Therefore, if we compose one configuration with another that has an actor with a nonterminating computation, computation in the first configuration may nevertheless proceed as before.

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \;\Rightarrow\; \langle\, \alpha, \texttt{[}e\texttt{]}_a \;\big|\; M \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, \texttt{[}e'\texttt{]}_a \;\big|\; M \,\rangle^{\rho}_{\chi}$$

$$\langle\, \alpha, \texttt{[}R[\texttt{newactor(e)}]\texttt{]}_{\texttt{a}} \;\big|\; M \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, \texttt{[}R[a]\texttt{]}_a, \texttt{[}e\texttt{]}_{a'} \;\big|\; M \,\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

$$\langle\, \alpha, \texttt{[}R[\texttt{send}(a, v_1)]\texttt{]}_a \;\big|\; M \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, \texttt{[}R[\texttt{nil}]\texttt{]}_a \;\big|\; M, \mathit{msg} \,\rangle^{\rho}_{\chi} \qquad \mathit{msg} = \langle a \Leftarrow v_1 \rangle$$

$$\langle\, \alpha, \texttt{[}R[\texttt{ready}(v)]\texttt{]}_{\texttt{a}} \;\big|\; \langle a \Leftarrow cv \rangle, M \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, \texttt{[}\texttt{app}(v, cv)\texttt{]}_a \;\big|\; M \,\rangle^{\rho}_{\chi}$$

$$\langle\, \alpha \;\big|\; M, \mathit{msg} \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \;\big|\; M \,\rangle^{\rho'}_{\chi} \qquad \text{if } \mathit{msg} = \langle a \Leftarrow cv \rangle,\ a \in \chi, \text{ and } \rho' = \rho \cup (\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha))$$

$$\langle\, \alpha \;\big|\; M \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \;\big|\; M, \mathit{msg} \,\rangle^{\rho}_{\chi\cup(\mathrm{FV}(cv)-\mathrm{Dom}(\alpha))} \quad \text{if } \mathit{msg} = \langle a \Leftarrow cv \rangle,\ a \in \rho \text{ and } \mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

**Figure 2. Actor transitions.**

Equivalence is a fundamental property that is often used in reasoning about programs. Specifically, equivalences between actor expressions and actor configurations are significant. The notion of equivalence in our theory is a kind of observational equivalence that is defined by adding an observable distinguished *event* to the set of transitions. This technique is a variant of operational equivalence as defined in [15]. Two actor expressions may be plugged into a context to see if the event occurs in one or the other case. Two expressions are considered equivalent if they have the same observations over all possible contexts.

Equivalence laws and proof techniques to simplify reasoning about actor systems have been developed. In particular, the proof techniques allow us to use canonical multi-step transitions as well as to reduce the number of contexts that need to be considered in studying the equivalence of systems. The algebraic properties of equivalence laws and the compositionality of configurations allows algebraic proof techniques to be used as well. A more abstract model of actors has been developed in [22].

A concrete way to think of actors is that they represent an abstraction over concurrent architectures. An actor runtime system provides the interface to services such as global addressing, memory management, fair scheduling, and communication. It turns out that these services can be efficiently implemented, thus raising the level of abstraction while reducing the size and complexity of code on concurrent architectures [11]. Alternately, such services can be implemented by developing libraries in traditional programming languages.

## 3 Meta-Level Architectures and Programming Abstractions

A number of *meta-level models* have been devised to manage the complexity of large systems. Such models support a separation of functional design concerns from non-functional ones (e.g. see [24]). In these models, a meta-architecture allows components to be placed into *base* and *meta* layers. *Base-level* components capture the functional aspects of an application, while other aspects of the system such as synchronization, security, fault-tolerance, and coordination of base-level objects are performed by *meta-level* components. New meta-levels can be recursively imposed on each meta-level and therefore the same flexibility in separating concerns can be provided for each level.

Meta-level architectures have been used to support a number of application areas [5, 21, 8]. For example, Astley and Agha define for instance new abstractions called *components*, *connectors*, and *actor groups* [6] (see also [5]). In this model, an actor group represents an encapsulation boundary, which protects internal actors from external interactions: actors within a group may only exchange messages with other actors in the same group. Composition operators are used to build connections between groups. These composition operators are also used to install meta-level customizations on group actors.

Programming abstractions can be built in terms of a meta-level architecture and such abstractions allow a more declarative implementation of different kinds of non-functional requirements in open distributed systems. For example, Frolund and Agha introduce meta-level abstractions to coordinate the interaction among base-actors [10, 8]. In this work, a programming construct called *synchronizers* is defined to coordinate the interactions of a group of actors. A variant of synchronizers has been used to represent real-time constraints [16, 12]. Other meta-level abstractions developed at the University of Illinois Open Systems Laboratory provide abstractions to represent protocols that can be dynamically installed [21, 5]. In particular, a number of fault-tolerance protocols have been expressed and implemented in this way.

Adaptability is an important property of dependable systems and can be obtained through the use of meta-level frameworks. Adaptively dependable systems function for long durations despite changing execution environments. An effective mechanism to achieve adaptability is dynamic installation of protocols. Dynamic protocol installation can be used in combination with exception handling to build robust applications with respect to faults and unpredicted changes. Sturman and Astley use meta-level frameworks to implement dynamic protocol installation [21, 5].

In the following section we illustrate an actor-based meta-framework and its application in designing dependability protocols.

## A Meta-Architectural framework for Actors

The event meta-architecture, developed in [6, 5], allows the modification of the basic semantics of actors to enable transparent addition of new services such as dependability services. To do this we extend the basic actor model by a meta layer, modeled as actor computations, and a customized protocol which links the base application and meta layers. In this subsection we explain basic concepts and mechanisms of the extended model.

### Inter-Level Interaction

To support the addition of transparent meta-level services, we need mechanisms to notify meta-level components whenever relevant actions take place at the base-level. These mechanisms are based on several of primitives. The first primitive, which is the main constituent of semantics of concurrent systems, is the notion of *event*. An event represents some state change in the system. Events, together with a partial-order causal relation, define the semantics of an actor computation. In an actor system, there are three important events : *message send*, *actor creation*, and *request for next message* (performed by the `ready()` primitive). In other smeantic models, other events can be defined.

The model of computation used for the meta-level determines the nature of the protocol for notifying meta-level components whenever base-level events occur. In our framework, meta-level computation is performed by actors. A natural solution to the problem of informing meta-actors entails sending them messages with descriptions of base-level events. A *signal* is the means through which base-level events create meta-messages. For each type of event, we define a signal and the format and contents of associated meta-messages. Whenever a base-level event occurs, the corresponding signal will generate a meta-message. An important design decision is to determine which meta-actor should receive the generated message. Figure 3 lists the typical signals and notifications defined in the Astley's framework.

| Actor Transitions | | |
|---|---|---|
| Event | | Behavior |
| Signals | **transmit**($msg$) | Triggered when a base-actor executes a `send(`$m$`)` operation. Base actor blocks until a **continue** is received. The argument $msg$ is a message structure which encapsulates the destination, method to invoke, and arguments of the message (that is, $m$). The default system behavior is to send the message and send a **continue** notification to the signaling actor. |
| | **ready**() | Triggered when a base-actor requests the next available message via a `ready()` operation. The default system behavior is to get the next available message and deliver it to the actor by generating a **deliver** notification. |
| | **create**($beh$) | Triggered when a base-actor executes a `create(`$beh$`)` operation. Base-actor blocks until a **newActor** is received. The argument $beh$ is the behavior of the new actor to be created. The default system behavior is to create the new actor and deliver its address to the base-actor via a **newActor** notification. |
| Notifications | **continue**() | Resumes a base-actor blocked on a `send(`$m$`)` operation. |
| | **deliver**($msg$) | Delivers a message to a base-actor. The argument $msg$ is a message structure indicating the method and arguments to invoke on the resumed actor. |
| | **newActor**($a$) | Returns the address of a newly created actor to a base-actor blocked on a `create(`$beh$`)` operation. The argument $a$ is the address of the newly created actor. |

**Figure 3. Each signal has a default behavior corresponding to the actor semantics of the associated operation. (From [5])**

The events we are concerned with here are all atomic; in that their executions never overlap in time. Atomicity is an important property in this case – it prevents interference between semantics of different events. To guarantee atomicity in our framework, we block base-level computation whenever a signal is generated. When a meta-level actor finishes its computation, it notifies the blocked base-level actor to resume its computation. Such notifications release base-level computation and transfer information that is needed by the resuming event (see Figure 4).

In several models there is one meta-actor for every base-actor (e.g. [5, 24]). However, other models are possible. For example, Yonezawa's group has used a model with shared meta-object [25]. Composition of protocols is achieved by stacking events and notifications, that is, a meta-actor is customized by a meta-meta-actor.

### Example: Transparent Encryption

So far we have seen that in our framework we can add a meta-actor for every base-actor. The meta-actor has one method for each type of signal. Here we illustrate the signal-notify

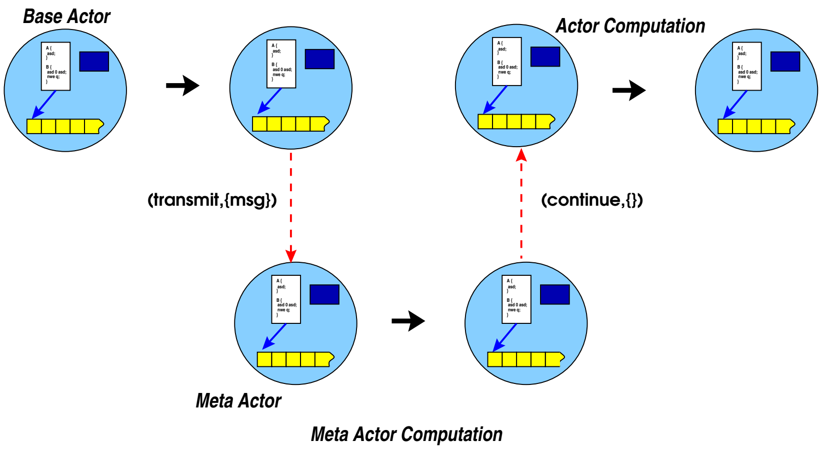


**Figure 4. Inter-Level Interaction: A *meta-actor* customizes a *base-actor* by intercepting signals. The meta-actor generates a notification after each signal has been processed and frees the blocked base-actor.**

framework by a simple example that shows how an encryption scheme can be transparently added to actor communications.

```
actor Encrypt (actor receiver) {
    // Encrypt outgoing messages if
    // they are targeted to the receiver
    method transmit(Msg msg) {
        actor target = msg.dest;
        if (target == receiver)
            target ← encrypt(msg);
        else
            target ← msg;
        continue();
    }
}
```

```
actor Decrypt() {
    // Decrypt messages targeted
    // for base actor (if necessary)
    method ready() {
        msg = Scheduler.NextMsg(
            baseactor);
        if (encrypted(msg))
            deliver(decrypt(msg));
        else
            deliver(msg);
    }
}
```

**Figure 5. Meta-Level Implementation of Encryption: The `Encrypt` meta-actor intercepts `transmit` signals and encrypts outgoing messages. The `Decrypt` policy actor intercepts messages targeted for the receiver (via the `ready` method) and, if necessary, decrypts an incoming message before delivering it (From [5].**

Figure 5 defines two meta-actors written in high level code . These actors add an encryption service to base-level computation. One meta-actor, `Encrypt`, listens to messages sent by the corresponding base-actor to a certain destination. It has only one method, `transmit()`, which is invoked via the signal mechanism whenever the base-actor sends a message. `transmit()` then verifies the destination of the message and if it matches the intended receiver, it encrypts the message using some scheme. Otherwise it simply delivers the original message.

The second meta-actor, `Decrypt`, listens to `ready()` messages caused by ready instructions executed by base actors. In this example, we suppose the invocation is performed after the scheduler chooses a message to be delivered. That message is passed to `Decrypt` as an argument of `ready()`. The meta-actor then the message if necessary and delivers the result to the receiving base-actor.

### 3.0.1 Example: Transparent Implementation of Primary-Backup Protocol

Using the signal-notify framework, a simple replication scheme based on the *primary-backup* protocol can be implemented. Figure 6 shows a `Replicator` and a `Backup` actor. `Replicator` perceives signals from a base-actor, which must be replicated. For every message delivered to the base-actor, `Replicator` sends a stamped copy to `Backup`, which saves it. In case of failure, `Backup` will contain a history of state snapshots that can be used for recovery.

## 4 Verification of Security Protocols

A significant problem in the design of secure distributed systems is authentication of communicating principals and distribution of session keys. Recently, several models and frameworks have been proposed to study and analyze authentication protocols and to verify their safety under various types of attacks. The bases of these frameworks vary from logics, such as belief logics[7] and inductive methods [14], to semantic ones, such as CSP [17] and CCS [1].

The Actor model provides a semantic approach to specifying and reasoning about security protocols which has some potential advantages. In this approach, attackers and possible threats are expressed in terms of actor configurations containing malicious attackers modeled as actors. These configurations are composed with a configuration containing actors that implement a protocol, and the effects of the interaction that results from this composition are analyzed in order to verify the security of the protocol.

An advantage of using the Actor model is that it can naturally express asynchrony and concurrency – this allows specification and analysis of overlapping executions of protocol instances something that has not yet been addressed in other modeling frameworks. Moreover, because a protocol is modeled as an open configuration, one can, in principle, model and analyze general schemes of attacks as arbitrary configurations that can be composed with the protocol.

The motivating idea is to verify that an application using a security protocol expresses the same behavior in a threatless environment as in all possible environments in which

```
actor Replicator(actor backup) {
    int processed = 0;
    int count = 0;
    boolean waiting = false;
    Queue mailQ;

    // Copy incoming messages to backup
    method rcv(Msg m) {
        // Send a stamped message to the backup
        backup ← rcvMsg(m, count++);

        // Queue until our base actor is ready
        if (waiting) {
                waiting = false;
                deliver(m);
        } else
                mailQ.enqueue(m);
    }

    // Forward state to backup and
    // deliver next message
    method ready(State s) {
        backup ← rcvState(s, processed++);

        if (!mailQ.empty())
                deliver(mailQ.dequeue());
        else
                waiting=true;
    }
}
```

```
actor Backup() {
    int count;
    State last;
    PriorityQueue unprocessed;

    // Receive new unprocessed message
    method rcvMsg(Msg m, int seq) {
        unprocessed.enqueue(m, seq);
    }

    // Receive new state
    method rcvState(State s, int seq) {
        last = s;

        Remove all message in "unprocessed"
        with sequence number less than seq
    }
}
```

**Figure 6. Meta-Level Implementation of Replication: An instance of `Replicator` is installed on the actor to be replicated. An instance of `Backup` receives state snapshots from the `Replicator` so that it can assume the role of the replicated actor if a failure occurs (From [5]).**

an arbitrary attacker might threaten the security of the application. In this section, we briefly describe the work of Skalka and Smith [18], who have analyzed security properties of the NSPK protocol under specific types of attack by formalizing them using Specdiag, a graphical specification language for actors [19]. It should be observed that research is currently in progress to find proofs of security by making weaker assumption about the nature of attackers.

Skalka and Smith have specified the NSPK authentication protocol in Specdiag. Different classes of attackers are modeled as actors or configurations of actors. The reasoning uses a notion of observational equivalence defined in terms of interaction paths [22].

The NSPK protocol as analyzed in Skalka and Smith's paper is defined as follows:

$$\begin{aligned} A &\rightarrow B : p_b(n_a.A) \\ B &\rightarrow A : p_a(n_a.n_b.B) \\ A &\rightarrow B : p_b(n_b) \end{aligned}$$

where $X \rightarrow Y : m$ means that $X$ sends message $m$ to $Y$, $p_a$ and $p_b$ are public keys for $A$ and $B$ respectively, and $x.y$ is used for concatenation of message contents $x$ and $y$. $p_x(m)$ refers to the encoded message using $X$'s public key. Only $X$ can decode the message using its private key. $n_a$ and $n_b$ refer to *nonces* generated by $A$ and $B$ respectively (nonces can be thought of as freshly generated identifiers). For theoretical analysis, it is generally assumed that nonces cannot be guessed.

The protocol starts when $A$ sends a nonce and its identity to $B$. $B$ replies with corresponding information about itself, plus $A$'s nonce. $A$ can verify $B$'s identity and be certain the received message was in reply to its own message and not a replay of a past communication. By sending $B$'s nonce back to $B$, $A$ similarly assures $B$ of the validity of $A$'s identity.

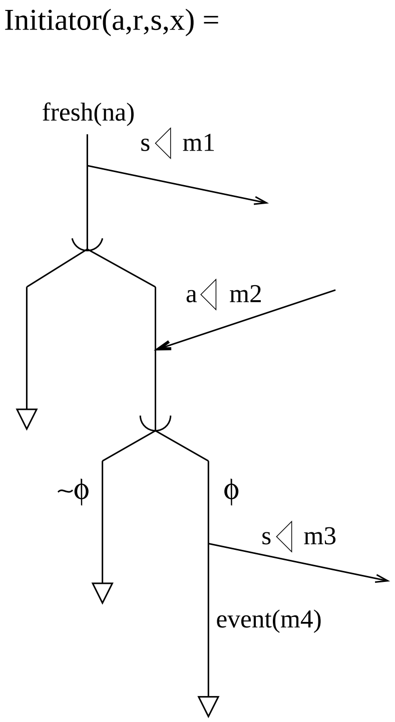
$A$ (the *initiator* of the protocol) and $B$ (the *responder*) are modeled as actors that interact via asynchronous messages (see Figure 7 and Figure 8). We must also model an environment in which malicious attackers may try to impose threats on the protocol by intercepting messages, replaying messages, or disguising themselves as one of the principals involved in the protocol.

The medium through which the initiator and the responder communicate is explicitly modeled as an actor called *router* (see Figure 9). The router can be replaced by *enemy* actor(s) that simulate various attackers and, as a result, one can analyze their effect on the protocol by verifying the interaction paths of *initiator* and *responder*. The diagrams for "adversary" actors are not shown here, but Skalka and Smith argue that they are not needed for the verification process because the sort of messages these adversaries send is all that defines the attack.

Skalka and Smith use a proof technique to overcome this difficulty. The idea is to identify the main characteristics of the enemy configuration and devise a simpler diagram that divides the original diagram into more intuitive parts. The motivating idea here is that proofs may become easier using intuitive visual representation. However, this step is by no means automated – it requires rigorous understanding of the nature of attackers' behavior.

The next step of the proof is to show the equivalence of the obtained diagram to the original one. Skalka and Smith give a semantic proof of this equivalence by using the interaction paths semantics of Talcott [22].

Because Actors model open distributed systems, sequentiality and closed-world assumptions can be relaxed. For example, one can model the possibility of concurrent sessions. Future research to explore the implications of this is on-going.

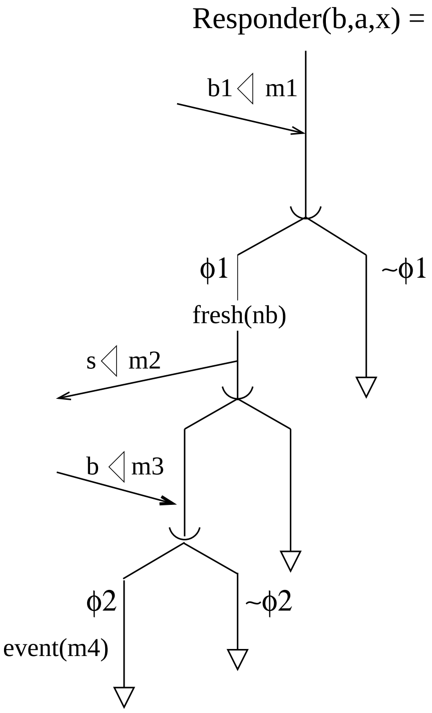


$$\begin{aligned} m1 &= \lfloor n_a.a \rfloor_r \\ \phi &\equiv m2 = \lfloor n_a.n_r.r \rfloor_a \\ m3 &= \lfloor n_r \rfloor_r \\ m4 &= succ(a, r, n_a, n_r) \end{aligned}$$

**Figure 7.** *Initiator* **with name a (From [18]). The initiator sends an encrypted copy of its nonce to the other principal. If the reply contains intiator's nonce, a reply will be sent to confirm the recpetion of the other principal's nonce. The initiator generates an event announcing values that indicate a valid termination of this principal's behavior. Only pahts that generate a correct event are considered to guarantee security of the protocol. (All messages are encrypted with the public key of the receiver.)**

## 5 Discussion

A primary advantage of a meta-level approach to software design is its provision for dynamic and transparent addition and modification of non-functional services to the system. Nevertheless, when adding meta-level objects, one must be careful not to interfere with



$$\begin{aligned}\phi_1 &\equiv m1 = \lfloor n_i.i \rfloor_i \\ m2 &= \lfloor n_i.n_b.b \rfloor_i \\ \phi_2 &\equiv m3 = \lfloor n_b \rfloor_b \\ m4 &= succ(b, i, n_b, n_i)\end{aligned}$$

**Figure 8.** ***Responder* with name b (From [18]). The responder receives the initiator's nonce and decrypts it using its private key. $\phi_1$ indicates the paths taken if the message is correctly decrypted. The responder will then generate a nonce and send it back to the initiator. After receiving the initiator's final response, it generates a success event if the last response is from the initiator and belong to the current session.**

intended domain-specific behavior of the base-level components, or at least be able to control interferences if intended.

Much research in this area remains to be done. Some early work has been carried out by Venkatasubramanian and Talcott [24] who develop a two-level meta-level model of open distributed system. The TLAM model(Two Level Actor Machine) provides a formal semantics for such systems and therefore a basis for specifying and reasoning about properties of and interactions between components.

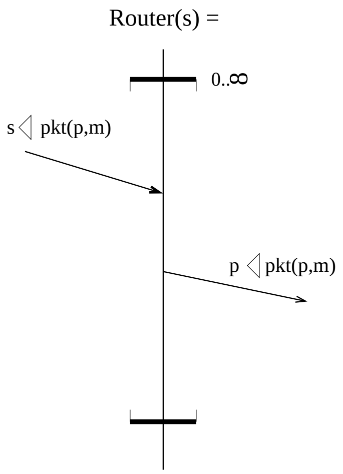


**Figure 9.** $\mathit{Router}$ **with name s (From [18]). Principals send their messages to the router, which then forwards them to the intended destinations. The router repeats this cycle forever.**

Several kinds of interference can be observed in an actor-based meta-level framework. Potentially, interferene may occur between actors with a common acquaintance, or even between actors lying at different levels. Moreover, different meta-level actors may implement different services and cause unpredictable modification of base-actor interactions. Making non-interference properties explicit has the advantage of making specifications modular and composable.

The work on security protocols that has been presented here can be extended in another direction, namely, showing how the protocol may be embedded in a meta-level component and its behavior composed with arbitrary applications with security requirements. Moreover, newer generation of distributed systems are commonly implemented using protocol stacks. This can be modelled by the meta-level approach. By contrast, traditional analysis techniques do not provide methods to drill down into the implementation layers of the underlying network and operating system. Modelling such interaction in a compositional way will allow reasoning about protocol interactions. This research will lead to a formulatation of proof rules that simplify reasoning about security protocols in an open, concurrent systems which has fault-tolerant protocols embedded in it. At the same time, the meta-architecture will provide a modular implementation of protocols and applications – going a long way towards addressing the difficulty of software maintenance.

## Acknowledgements

We'd like to thank past and present members of the Open Systems Laboratory whose research continues to provide an inspiration to our current work. In addition, we would like to thank Carlos Varela and James Waldby for comments on this paper. The research described in this paper has been made possible in part by NSA contract MDA904-98-C-A081 and the Air Force Office of Science Research under F49620-97-1-0382.

## References

[1] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1993.

[3] Gul Agha. *Actors: A Model of Concurrent Computation.* MIT Press, 1986.

[4] Gul Agha, Svend Frølund, Wooyoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology*, May 1993.

[5] Mark Astley. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures.* PhD thesis, University of Illinois at Urbana-Champaign, May 1999.

[6] Mark Astley and Gul Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. *Sixth International Symposium on the Foundations of Software Engineering ACM SIGSOFT*, 23(6):1–9, November 1998.

[7] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:18–36, February 1990.

[8] Svend Frølund. *Constraint-Based Synchronization of Distributed Activities.* PhD thesis, Department of Computer Science, University of Illinois at Urbana Champaign, 1994.

[9] Svend Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.

[10] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993.* Springer Verlag, 1993. LNCS 707.

[11] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Supercomputing '95.* IEEE, 1995.

[12] B. Nielsen and sc G. Agha. Towards reusable real-time objects. In *Annals of Software Engineering: Special Volume on Real-Time Software Engineering.* to be published.

[13] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment. Available for download at http://www-osl.cs.uiuc.edu/foundry.

[14] L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Computer Security*, 6:85–128, 1998.

[15] G. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[16] Shangping Ren. *An Actor-Based Framework for Real-Time Coordination.* PhD thesis, Department Computer Science. University of Illinois at Urbana-Champaign, 1997.

[17] Steve Schneider. Modelling Security Properties with CSP. Technical report, Royal Holloway Technical Report CSD-TR-96-04, 1996.

[18] C. Skalka and S. Smith. Verifying security protocols with specification diagrams. Submitted for publication.

[19] Scott Smith. On specification diagrams for actor systems. In C. Talcott, editor, *Proceedings of the Second Workshop on Higher-Order Techniques in Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998. (to appear).

[20] Daniel C. Sturman. Fault-adaptation for systems in unpredictable environments. Master's thesis, University of Illinois at Urbana-Champaign, January 1994.

[21] Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[22] C. L. Talcott. Interaction semantics for components of distributed systems. In *FMOODS'96*, 1996.

[23] C. Varela and G. Agha. A Hierarchical Model for Coordination of Concurrent Activities. In P. Ciancarini and A. Wolf, editors, *Third International Conference on Coordination Languages and Models (COORDINATION '99)*, number 1594 in LNCS, pages 166–182. Springer-Verlag, April 1999. http://osl.cs.uiuc.edu/Papers/Coordination99.ps.

[24] N. Venkatasubramanian and C. L. Talcott. Reasoning about Meta-Level Activities in Open Distributed Systems. In *Principles of Distributed Computing*, 1995.

[25] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*, chapter Reflection in an Object-Oriented Concurrent Language, pages 45–70. MIT Press, Cambridge, Mass., 1990.