A Hierarchical Model for Coordination of Concurrent Activities

Carlos Varela and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
cvarela@uiuc.edu, agha@cs.uiuc.edu
http://osl.cs.uiuc.edu

Abstract. We describe a hierarchical model for coordination of concurrent activities based on grouping actors into *casts* and coordinating casts by actors that are designated *directors*. The hierarchical model provides a simple, intuitive basis for actor communication and coordination. Casts serve as abstraction units for naming, migration, synchronization and load balancing. *Messengers* are actors used to send messages with special behaviour across casts. Moreover, an implementation of the hierarchical model does not require a reflective run-time architecture. We present the operational semantics for our model and illustrate the model by two sample applications: an atomic multicast protocol and a messenger carrying remote exception-handling code. These applications have been implemented in Java, leveraging the existence of cross-platform, safe virtual machine implementations.

1 Motivation

In order to address the difficulty of developing and maintaining increasingly complex software systems, a number of methodologies to facilitate a separation of design concerns have been proposed. A number of researchers, including the authors, have argued for a separation of code implementing the functionality of software components (the *how*) from code for "non-functional" properties, such as the relation between otherwise independent events at different actors (i.e. the *when*) [3, 15, 38], and the mapping of actors on a particular concurrent architecture (the *where*) [35]. Such separation of concerns results in modular code which allows software developers to reason compositionally and to reuse code implementing functionality, coordination, placement policies, and so forth independently.

We model coordination hierarchically by grouping actors into *casts*. Each cast is coordinated by a single actor called its *director*. We also introduce migrating *messenger* actors to facilitate remote cast-to-cast communication. Casts serve as abstraction units for naming, migration, synchronization and load balancing. Messengers are migrating actors used to send messages (possibly including behaviours) across casts. Our model explicitly represents locality in order to enable an explicit specification of the structure of the information flow between remotely located casts using actor migration.

The hierarchical model is motivated by social organizations, where groups and hierarchical structures allow for effective information flow and coordination of activities. The basic idea is not entirely new. For example, Simon [37] proposed hierarchies as an appropriate model to represent different kinds of complex systems – viz., social, biological and physical systems as well as systems carrying out symbolic computations. This proposal was based on the observation that interactions between components at different levels in a hierarchy are often orders of magnitude smaller than interactions within the sub-components. Simon argues that the near-decomposability property of hierarchic systems also allows for their natural evolution and illustrates this argument with examples drawn from systems such as multi-cellular organisms, social organizations, and astronomical systems.

Regardless of the philosophic merits of Simon's proposition, we believe that a hierarchical organization of actors for coordination is not only fairly intuitive to use but quite natural in many contexts. By using a hierarchical model, we simplify the description of complex systems by repeatedly subdividing them into components and their interactions.

A number of models of coordination require reflective capabilities in the implementation architecture; such reflection is used to support meta-level actors that can intercept and control base-level actors (e.g., see [32, 38, 42, 44]). The need for reflection creates two difficulties. First, it requires a specialized runtime system. Second, it complicates the semantics: correctness of a particular application not only depends on the semantics of application level actors, but also on the semantics of the meta-level architecture.

The hierarchical model proposed in this paper only uses abstractions that are themselves first-class actors. In particular, there is no requirement for meta-level actors which are able to intercept base-level actor messages. The cost of this simplicity results in at least two major disadvantages for the hierarchical model. First, the model does not support the degree of transparency that can be afforded by defining coordination abstractions using reflective architectures. Second, because the hierarchical model is limited to customizing communication, it is not as flexible as a reflective model. However, observe that the actor model represents all coordination through its message-passing semantics; thus customizing communication is far more powerful than it may first appear to be.

Despite these limitations, it is our conjecture that for a large number of applications, the simplicity of the hierarchical model suffices.

The outline of the rest of the paper is as follows. Section 2 briefly discusses research on Actors and other related topics. Section 3 gives an informal overview of the hierarchical model of actor communication and coordination. Section 4 presents the operational semantics of the model based on actor configurations. Section 5 shows two basic but powerful applications of the model: atomic multicasting and remote exception handling via messengers. Finally, the last section concludes with a discussion including future directions.

2 Related Work

Actors [1, 19] extend sequential objects by encapsulating a thread of control along with procedures and data in the same entity; thus actors provide a unit of abstraction and distribution in concurrency. Actors communicate by asynchronous message passing. Moreover, message delivery is weakly fair – message delivery time is not bounded but messages are guaranteed to be eventually delivered. Unless specific coordination constraints are enforced, messages are received in some arbitrary order which may differ from the sending order. An implementation normally provides for messages to be buffered in a local mailbox and there is no guarantee that the messages will be processed in the same order as the order in which they are received.

The actor model and languages provide a very useful framework for understanding and developing open distributed systems. For example, among others applications, actor systems have been used for enterprise integration [39], real-time programming [36], fault-tolerance [4], and distributed artificial intelligence [13].

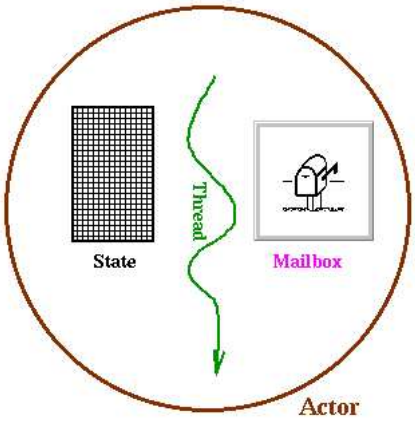

Fig. 1. Actors have their own thread of computation and mailbox

Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [26] and facilitates mobility [5]. However this flexibility comes at a price: given the autonomy of actors and the non-blocking nature of message-passing, coordinating actors is a difficult problem.

ActorSpaces [9] are computationally passive containers of actors. Messages may be sent to one or all members of a group defined by a destination pattern. *Casts* differ from ActorSpaces in that the management of messages coming to

a group is handled by a director, which is a computationally active component, and can therefore explicitly support multiple group messaging paradigms. On the other hand, the hierarchical model requires explicit description of how actors are grouped together, since policies such as pattern matching according to particular actor attributes are not directly supported.

Synchronizers [14, 15] are linguistic constructs that allow *declarative* control of coordination activity between multiple actors. Synchronizers allow two kinds of specifications: messages received by an actor may be disabled and, messages sent to different actors in a group may be atomically dispatched. These restrictions are specified using message patterns and may depend on the synchronizer's current state. *Directors* differ from Synchronizers in that directors are not declarative – rather a director is an actor: a director can receive messages and a director may itself be subject to coordination constraints. Coordination of directors not only allows for dynamic reconfigurability of coordination policies but also for a hierarchic composition of constraints.

The hierarchical model is more restrictive in that it requires actors to belong to a single cast at a given time. By contrast, the groups controlled by synchronizers may overlap arbitrarily. However, because the path for messages can be more rigid and the coordination more determinate in the hierarchical organization than in synchronizers, the former has the associated benefit of more predictable performance.

Communication-passing style (CmPS) [21] refers to a semantics in which data communication is always undertaken by migrating the continuation of the task requiring the data to the processor where the data resides. *Messengers* are similar to CmPS continuations in that they improve locality by eliminating coordination communication across machines. However, messengers are migrating first-class entities, or actors. Therefore, messengers can receive messages, change their state, persist over time, and support dynamic message delivery policies. On the other hand, messengers require the underlying run-time system to support actor migration.

Fukuda et al. [16] describe a paradigm for building distributed systems, named `Messengers`. The paradigm has similar motivation as our use of the messenger construct. However unlike Fakuda et al.'s work, we distinguish between actors and messengers (besides our use of hierarchies). Note that in early development of the actor model, Hewitt and his colleagues considered messages to be *unserialized* actors – i.e., messages (unlike serialized actors) could not change their local state (e.g., see [19]).

A number of coordination models and languages [11, 17] use a globally shared tuple space as a means of coordination and communication [10]. A predecessor of Linda, using pattern based data storage and retrieval was the Scientific Community Metaphor [27]. *Sprites*, the Scientific Community's computational agents, share a monotonically increasing knowledge base; on the other hand, Linda allows communication objects (tuples) to be removed from the tuple space. Additional work in this direction includes adding types to tuple spaces for safety, using objects instead of tuples and making tuple spaces first-class entities [20, 25, 31].

In contrast to these approaches, the hierarchical model of actor coordination that we present is based purely on communication which requires the sender to explicitly name the target of a message. Our use of the term coordination is different: following Wegner, we define coordination as *constrained interaction* [43] between actors. Such interaction does not require shared spaces.

The software architecture community has worked on architecture description languages (ADLs) to simplify the development of complex systems. Examples of these languages include Wright [7], Rapide [30] and DCL [8]. In DCL, components and connectors to describe a software architecture can be specified. Components represent the functional level, while connectors are lower-level abstractions which define how an architecture is deployed in a particular execution environment.

Much research and development in industry has focused on open distributed systems; some examples include CORBA [33] and Java-related [18] efforts (e.g., see RMI [22], JavaSpaces [23], JINI [41], and concurrency patterns [28]). Java transforms a heterogeneous network of machines into a homogeneous network of Java virtual machines, while CORBA solves the problems of interactions between heterogeneous environments [2]. While the hierarchical model's concept of actor casts and directors can be implemented using CORBA for coordination of concurrent activities, the concept of messengers is only realizable with approaches such as Java's virtual machine [29] that allow actor behaviours to be safely sent and interpreted across heterogeneous machines.

3 A Hierarchical Model for Coordination

Coordination in the hierarchical model is accomplished by constraining the receipt of messages that are destined for particular actors. An actor can only receive a message when the coordination constraints for such message receipt are satisfied. The coordination constraints are checked for conformance at special actors named *directors*. The group of actors coordinated by a director is defined as a *cast*. *Messengers* are special migrating actors that represent a message from a remote cast.

3.1 Directors and casts

Although the director of a cast plays special roles – such as the "coordinator" of actors internal to the cast, an interface to other casts, a request broker for the service provided by the cast, and so forth – a director has no more "power" than other actors that are in the cast. That is, as far as the run-time system is concerned, a director is just another actor. It is also possible to have completely "uncoordinated" actors, i.e. actors which do not belong to a cast and which therefore have no external constraints on message receipt and processing.

An actor can have at most one director at a given point in time. However, a director may itself belong to a cast and thus be coordinated by another director. This strategy allows for dynamic reconfigurability of coordination constraints,

as well as for modular constraint composition. The director-actor relationship forms a set of trees, named the *coordination forest*.

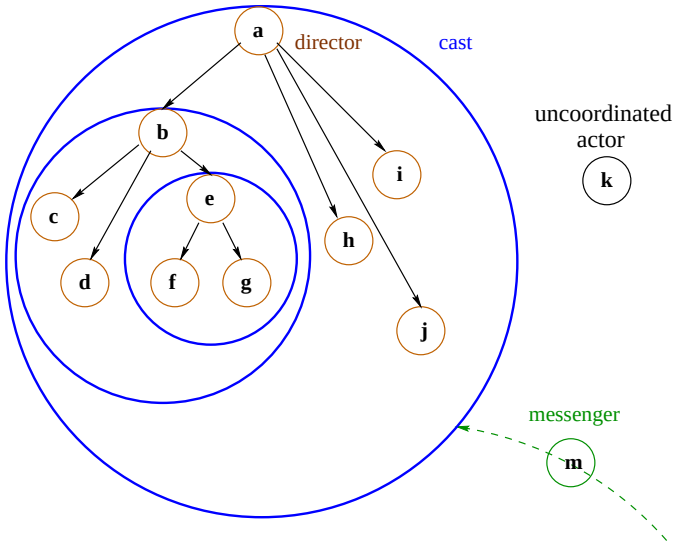

Fig. 2. Coordinated activity with *casts*, *directors* and *messengers*

A message from a `sender` actor is received by a `target` actor only after *approval* by all the directors in the target actor's coordination forest path up to the first common director, if such a director exists, and otherwise, approval is required of all directors in the `target`'s coordination forest path up to the top level.

Figure 2 shows a sample actor configuration. We illustrate valid message paths by describing a few examples based on this sample configuration:

- A message from any actor can go directly to actor `a`, or to actor `k`.
- A message from actor `f` to actor `g` has to go through their first common director, actor `e`.
- A message from `c` to `f` has to go through directors `b` and `e`. However a message from `f` to `c` only needs to be approved by `b`. This is because `e` is *not* in `c`'s coordination forest path.
- A message from `k` to `b,c,...j` has to be approved by `a`.

3.2 Messengers

A *messenger* is a special actor which migrates with the purpose of carrying a message from a local cast to a remote cast. A messenger may also possibly contain other actor behaviours. For example, messengers may:

- provide more robust message delivery by persisting over temporary failures in the target actor
- attempt message return upon failure in message delivery
- follow mobile actors
- acknowledge message receipt or processing
- handle common exceptions at the target actor's site
- be in charge of networking and naming issues.

4 Semantics based on Actor Configurations

We extend the operational semantics formulated by Agha, Mason, Smith and Talcott [6] in this section to capture the concept of casts and directors. Specifically, we add a δ function which maps actors to directors, and we tag messages to show that they require approval. We also remove the restriction that actor behaviour (λ-abstractions) may not be communicated.

The following two subsections introduce the language used and the reduction rules that define valid transitions between actor configurations.

4.1 A Simple Lambda Based Actor Language

Our actor language is a simple extension of the call-by-value lambda calculus with four primitives for creating and manipulating actors:

`newactor`(e) creates a new uncoordinated actor with behaviour e and returns its name.

`newdirectedactor`(e) creates a new actor with behaviour e, directed by the creator, and returns its name.

`send`(v_0, v_1) creates a new message with receiver v_0 and contents v_1 and puts the message into the message delivery system.

`ready`(v) signals the end of the current computation and the ability for the actor to receive a new message, with behaviour v.

For more details on this basic actor language, we refer the reader to [6].

4.2 Operational Semantics for Coordinated Configurations

In this subsection, we give the semantics of actor expressions by defining a transition on coordinated open configurations.

We take as given countable sets At (atoms) and X (variables).

Definition (V E M): The set of *values*, V, the set of *expressions*, E, and the set of *messages*, M, are defined inductively as follows:

$$\mathsf{V} = \mathsf{At} \cup \mathsf{X} \cup \lambda\mathsf{X}.\mathsf{E} \cup \mathtt{pr}(\mathsf{V}, \mathsf{V})$$

$$\mathsf{E} = \mathsf{V} \cup \mathtt{app}(\mathsf{E}, \mathsf{E}) \cup \mathsf{F}_n(\mathsf{E}^n) \qquad \text{where } \mathsf{F}_n(\mathsf{E}^n) \text{ is all arity-n primitives.}$$

$$\mathsf{M} = \langle \mathsf{V} \Leftarrow \mathsf{V} \rangle_{\mathsf{X}}$$

We use variables for actor names. An actor can be either ready to accept a message, written $\texttt{ready}(v)$, where v is its behaviour, a lambda abstraction; or busy executing an expression, written e. A message from a source actor a targeted to an actor with name v_0, and contents v_1 is written $\texttt{<}v_0 \Leftarrow v_1\texttt{>}_a$.

Let $\mathbf{P}_\omega[\mathsf{X}]$ be the set of finite subsets of X, $\mathbf{M}_\omega[\mathsf{M}]$ be the set of (finite) multi-sets with elements in M, $\mathsf{X}_0 \xrightarrow{f} \mathsf{X}_1$ be the set of finite maps from X_0 to X_1, $\mathrm{Dom}(f)$ be the domain of f and $\mathrm{FV}(e)$ be the set of free variables in e. We define actor configurations as follows.

Definition (Actor Configurations (K)): An *actor configuration* with director map, δ, actor map, α, multi-set of messages, μ, receptionists, ρ, and external actors, χ, is written

$$\langle\, \delta \;\big|\; \alpha \;\big|\; \mu \,\rangle^{\rho}_{\chi}$$

where $\rho, \chi \in \mathbf{P}_\omega[\mathsf{X}]$, $\delta \in \mathsf{X} \xrightarrow{f} \mathsf{X}$, $\alpha \in \mathsf{X} \xrightarrow{f} \mathsf{E}$, $\mu \in \mathbf{M}_\omega[\mathsf{M}]$, and let $A = \mathrm{Dom}(\alpha)$ and $D = \mathrm{Dom}(\delta)$, then:

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

(1) if $a \in A$, then $\mathrm{FV}(\alpha(a)) \subseteq A \cup \chi$, and if $\texttt{<}v_0 \Leftarrow v_1\texttt{>}_a \in \mu$ then $\mathrm{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

(2) if $a \in D$, and $a_0 = \delta(a)$, $a_1 = \delta(a_0)$, $a_2 = \delta(a_1)$, ..., $a_n = \delta(a_{n-1})$, such that $a_n \notin D$, then $\forall i \in 0..n : (a_i \in A \text{ and } a_i \neq a)$.

The last rule restricts actor configurations so that (1) all directors in the coordination forest path for an actor belong to the same configuration, and (2) no cycles are allowed in the actor-director relationship.

Definition (Coordination Forest Paths (Δ)): The coordination forest path of an actor a, $\Delta(a)$, in a configuration κ with director map δ, is defined as:

$$\Delta(a) = \begin{cases} \{a\} & \text{if } a \notin \mathrm{Dom}(\delta) \\ \{a\} \cup \Delta(\delta(a)) & \text{otherwise} \end{cases}$$

To describe the internal transitions between configurations other than message receipt, an expression is decomposed into a reduction context filled with a redex. For a formal definition of reduction contexts, expressions with a unique hole, we refer the reader to [6]. Suffice it to say here that R in the following definition, ranges over the set of reduction contexts. The purely functional redexes inherit the operational semantics from the purely functional fragment of our actor language. The actor redexes are: $\texttt{newactor}(e)$, $\texttt{newdirectedactor}(e)$, $\texttt{send}(v_0, v_1)$, and $\texttt{ready}(v)$.

Definition ($\mapsto$): The single-step transition relation $\mapsto$, on actor configurations is the least relation satisfying the following conditions: [1]

<fun : a>

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \langle\, \delta \mid \alpha\{a \to e\} \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \delta \mid \alpha\{a \to e'\} \mid \mu \,\rangle^{\rho}_{\chi}$$

<newactor : a, a'>

$$\langle\, \delta \mid \alpha\{a \to R[\mathtt{newactor}(e)]\} \mid \mu \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \delta \mid \alpha\{a \to R[a'], a' \to e\} \mid \mu \,\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

<newdirectedactor : a, a'>

$$\langle\, \delta \mid \alpha\{a \to R[\mathtt{newdirectedactor}(e)]\} \mid \mu \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \delta\{a' \to a\} \mid \alpha\{a \to R[a'], a' \to e\} \mid \mu \,\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

<send : a, v_0, v_1>

$$\langle\, \delta \mid \alpha\{a \to R[\mathtt{send}(v_0, v_1)]\} \mid \mu \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \delta \mid \alpha\{a \to R[\mathtt{nil}]\} \mid \mu, \langle v_0 \Leftarrow v_1 \rangle_a \,\rangle^{\rho}_{\chi}$$

<redirect : a, v_0, v_1>

$$\langle\, \delta \mid \alpha \mid \mu, \langle v_0 \Leftarrow v_1 \rangle_a \,\rangle^{\rho}_{\chi} \mapsto \langle\, \delta \mid \alpha \mid \mu, \langle \delta(v_0) \Leftarrow \mathtt{msg}(v_0, v_1) \rangle_a \,\rangle^{\rho}_{\chi}$$
if $v_0 \in \mathrm{Dom}(\delta)$, $\delta(v_0) \neq a$, and $v_0 \notin \Delta(a)$

<receive : v_0, v_1>

$$\langle\, \delta \mid \alpha\{v_0 \to \mathtt{ready}(v)\} \mid \langle v_0 \Leftarrow v_1 \rangle_a, \mu \,\rangle^{\rho}_{\chi} \mapsto$$
$$\langle\, \delta \mid \alpha\{v_0 \to \mathtt{app}(v, v_1)\} \mid \mu \,\rangle^{\rho}_{\chi}$$
if $v_0 \notin \mathrm{Dom}(\delta)$, or $\delta(v_0) = a$, or $v_0 \in \Delta(a)$

<out : v_0, v_1>

$$\langle\, \delta \mid \alpha \mid \mu, \langle v_0 \Leftarrow v_1 \rangle_a \,\rangle^{\rho}_{\chi} \mapsto \langle\, \delta \mid \alpha \mid \mu \,\rangle^{\rho'}_{\chi}$$
if $v_0 \in \chi$, and $\rho' = \rho \cup (\mathrm{FV}(v_1) \cap \mathrm{Dom}(\alpha))$

<in : v_0, v_1>

$$\langle\, \delta \mid \alpha \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \delta \mid \alpha \mid \mu, \langle v_0 \Leftarrow v_1 \rangle_{a'} \,\rangle^{\rho}_{\chi \cup (\mathrm{FV}(v_1) - \mathrm{Dom}(\alpha))} \qquad a' \text{ fresh}$$
if $v_0 \in \rho$, $\mathrm{FV}(v_1) \cap \mathrm{Dom}(\alpha) \subseteq \rho$

The *redirect* and *receive* transition rules ensure that all directors up to the common ancestor between the sender and the target actors (or the highest di-

[1] For any function f, $f\{\mathrm{x} \to \mathrm{x}'\}$ is the function f' such that $\mathrm{Dom}(f') = \mathrm{Dom}(f) \cup \{\mathrm{x}\}$, $f'(\mathrm{y}) = f(\mathrm{y})$ for $\mathrm{y} \neq \mathrm{x}$, $\mathrm{y} \in \mathrm{Dom}(f)$, and $f'(\mathrm{x}) = \mathrm{x}'$. $\mathtt{msg}(v_0, v_1)$ is a value expression representing a message with target v_0 and contents v_1.

rector in the target's coordination forest path) get notified and control when to actually send a message to a target actor. Notice the locality of information flow: if two actors belong to the same cast, outside actors and directors, even if in their coordination hierarchy, need not be notified/interrupted. For the configuration sample in Figure 2, a message from `c` to `f`, $\texttt{<}f \Leftarrow v\texttt{>}_c$, is redirected to `e`, $\texttt{<}e \Leftarrow msg(f, v)\texttt{>}_c$, and subsequently to `b`, $\texttt{<}b \Leftarrow msg(e, msg(f, v))\texttt{>}_c$. After checking coordination constraints, `b` and `e` send it to the final target `f`. Director `a` is not involved in such internal coordination.

The last two transitions account for the openness of actor configurations. The first transition, *out*, represents a message delivered to an external actor. The second transition, *in*, represents a message coming from an outside system to one of the configuration's receptionists. Notice that this message will be redirected to the root of the receptionist's coordination forest path.

5 Sample Applications

In this section, we illustrate our model by presenting some simple applications of our model: namely, three atomic multicast protocols and an application of messengers to improve locality in remote exception handling.

5.1 Atomic Multicast Protocols

By an atomic multicast protocol, we mean that the receipt of a message by different members of a cast is such that, to an outside actor, the operation can be viewed as a single step. In other words, no other messages are received by any of the actors in the cast until all members of the group have received the message.

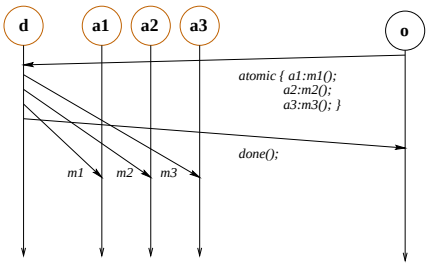

Fig. 3. Atomic multicast

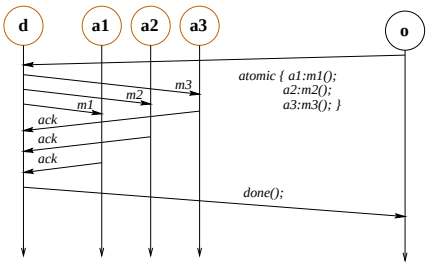

Fig. 4. Acknowledged atomic multicast

Figure 3 shows the trace of one possible execution of atomic multicasting. Vertical lines represent local time (increasing in the downward direction) and diagonal lines represent message traversals between actors. Even though the messages are originally directed to actors `a1..a3`, these actors are in the same cast,

and the cast's director can therefore send them atomically (without any interleaved messages) to the respective targets. Other messages sent at the same time to these actors are delayed until the atomic multicasting operation is finished. However, because the sending order may be different from the receiving order, this does not guarantee atomicity. A more robust implementation of atomic multicasting is as follows. A director waits for acknowledgement of message receipt by the coordinated actors, before notifying the outside actor of "finalization", as shown in figure 4.

If we wanted to go further and implement *group knowledge* [12] (i.e. everybody in the cast knows that everybody in the cast got the message), we could use a two-phase atomic multicasting protocol. In the first phase, we send the message to every actor in the cast; and in the second phase, we tell every actor in the cast that everybody acknowledged message receipt. Figure 5 shows a sample trace.

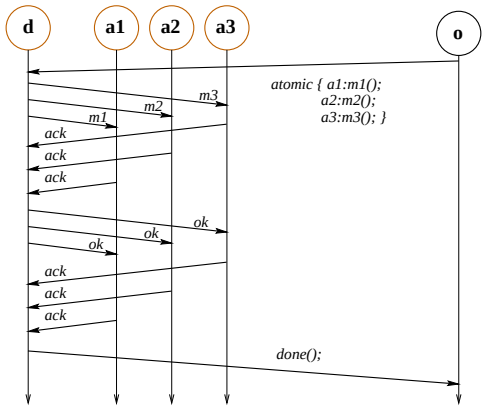

Fig. 5. Global knowledge atomic multicast

We have implemented these three atomic multicast protocols using the Actor Foundry 0.1.9 [34] for Solaris 2.5.1 with UDP plus a reliability transport protocol. Figure 6 shows the timing as measured from the outside actor, when all actors involved in the atomic multicast protocols resided in the same host. We used an empty message (no data) and we averaged the time taken over multiple executions. The results are as expected: time grows linearly with the number of actors, and the slope is doubled with reliability: it takes twice as many messages to do acknowledged atomic multicast than it takes to do basic atomic multicast, and it takes twice as many messages to do global knowledge atomic multicast than to do acknowledged atomic multicast.

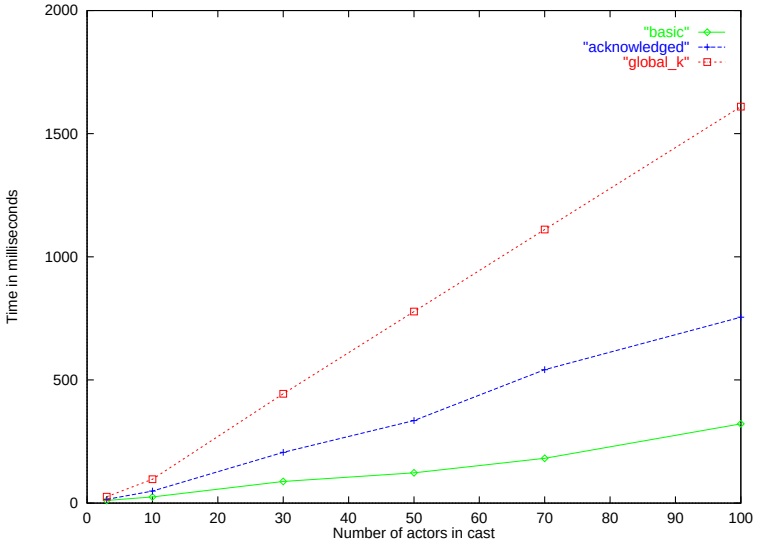

Fig. 6. Performance of different atomic multicast protocols (*in ms.*)

5.2 Remote Exception Handling with a Messenger

In our second example, we use a *messenger* to improve locality in exception handling across remotely located casts.

Suppose we have a `sender` actor that wants to send a `letter` to a `recipient` for processing. Furthermore, let's suppose that `sender` and `recipient` are in different hosts as shown in figure 7. If the recipient can't process the letter, an exception would be thrown from the recipient's host to the sender's host. After that, the sender may want to retry sending the letter at a later time.

We could improve locality using messengers by creating a new actor in host A and migrating it to host B with the specific behaviour of sending the original message and handling any associated exceptions that may arise remotely, as shown in figure 8.

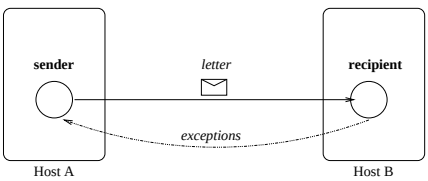

Fig. 7. Passive message across hosts

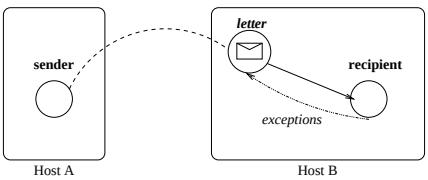

Fig. 8. `Messenger` across hosts

The algorithm for this example is given in Salsa, a language we're developing that extends Java to support actors and high-level coordination mechanisms. Salsa syntax is Java/C++-like with a few modifications to support actors [40]. In particular, `a1:m1->a2:m2` represents sending an asynchronous message `m1` to actor `a1` with customer `a2:m2`, meaning that `a1` should send another asynchronous message `m2` to the acquaintance actor `a2` after `m1` is processed. The return value of `m1` is passed as an argument to `a2:m2`.

The `sender` actor embodies the main program. It creates a recipient and a letter and sends the letter with three different exception handling protocols. The first one sends a passive message and ignores failures in the target actor. The second and third invocations use a messenger to handle exceptions at the target host appropriately (with methods `retry` and `failed` respectively). The sender also provides a method `log` in case the messenger ultimately fails to deliver the letter.

```
behaviour Sender {
  Letter letter;
  Recipient recipient;
  public void main(){
    /** Create letter messenger and recipient actor */
    letter = new Letter(self);
    recipient = new Recipient();
    /** Send a letter to a recipient, with possible failure */
    recipient:process(letter);
    /** Send a letter messenger that retries until successful */
    recipient:process(letter) -> letter:retry;
    /** Send a letter messenger with special exception handling */
    recipient:process(letter) -> letter:failed;
  }
  public void log(Recipient recipient){
    System.err:println(recipient + "failed processing the letter." );
  }
}
```

The recipient is an actor that provides a method to process a letter. Once a letter has been processed, it returns `null`. If the letter needs to be delegated to a third actor, such actor address is returned. If the actor can't temporarily process the letter, the method returns its own address for processing at a later time.

The `letter` messenger can either:

- retry forever until the letter is successfully processed, or
- retry once after one second and upon a new failure, send an exception message to the original sender to log an error.

```
behaviour Letter extends Messenger {
  Sender sender;
  /** Letter constructor */
  Letter(Sender s){
    sender = s;
  }
  /** Retries sending the letter until successfully processed */
  public void retry(Recipient recipient){
      if (recipient != null)
        recipient:process(self) -> self:retry;
    }
  /** Try once again after a second, else log an error */
  public void failed(Recipient recipient){
      if (recipient != null){
        System.wait(1000);
        recipient:process(self) -> self:failedTwice;
      }
  }
  /** Tell the sender of the letter that it couldn't be processed */
  public void failedTwice(Recipient recipient){
      if (recipient != null)
        sender:log(recipient);
  }
}
```

Notice that by reifying the passive letter into an actor, we're able to handle exceptions much more efficiently at the target actor's remote host.

6 Discussion

Traditional models for coordination of actors require the run-time system to support reflective capabilities in order to constrain certain types of messages from reaching their targets. A typical scenario is that of meta-level actors that are able to intercept message sending, receipt and processing for base-level actors.

The hierarchical model is less demanding of the run-time system in the sense that no special actor architecture is required for coordination of activities. Instead, actors are explicitly grouped in casts, which are in turn, coordinated by directors. These directors do not have more computational or communication capabilities than their coordinated counterparts. However, the model of communication ensures that directors are able to synchronize activities within their respective casts. Inter-cast communication can be performed with traditional asynchronous passive messages, or by using messengers.

A hierarchical structure of coordination may suggest possible bottlenecks at root directors in the coordination forest. However, because communications inside casts need not be coordinated by outside directors, the hierarchical model can be implemented efficiently. For directors above the common ancestor of two communicating actors, communications are internal transitions that need not interrupt outside actors.

Moreover, it is still possible to have actors that do not belong to any casts – such actors incur no unnecessary synchronization overhead at run-time. One strategy for avoiding potential bottlenecks – similar to the current one of replicating Internet domain name servers – is to co-locate directors on multiple nodes. This strategy would provide for limited cast state recovery in the case of failures in the cast's director.

Although the hierarchical model of actor coordination does not provide the sort of transparency and flexibility that reflective models can enable [24, 32, 38, 42, 44], we conjecture that despite its simplicity, it is powerful enough for many applications. The claims in this paper are tentative – further research on techniques for specifying and implementing the linguistic constructs in the hierarchical model is needed. We are currently also exploring the model's applicability to the problem of developing worldwide open distributed systems.

Acknowledgements

We'd like to thank past and present members of the Open Systems Laboratory who aided in this research. In particular, we'd like to express our gratitude to Mark Astley for developing the Actor Foundry and James Waldby for all those *yume* scripts. Thanks also to Reza Ziaei, Prassanna Thati and Nadeem Jamali for useful comments and discussions about this work. Last but not least, we'd like to thank Les Gasser and the anonymous referees for their comments.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] G. Agha, M. Astley, J. Sheikh, and C. Varela. Modular heterogeneous system development: A critical analysis of java. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 144–155. IEEE Computer Society, March 1998. http://osl.cs.uiuc.edu/Papers/HCW98.ps.

[3] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology*, May 1993.

[4] G. Agha, S. Frølund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III*, pages 345–363. International Federation of Information Processing Societies (IFIP), Elsevier Scienc Publisher, 1993.

[5] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In G. Weiss, editor, *Multiagent Systems: A Modern Approach to DAI.* MIT Press, 1999. To appear.

[6] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[7] R. Allen and D. Garlan. Formalizing architectural connection. In *International Conference on Software Enginnering*, pages 71–80. IEEE Computer Society, 1994.

[8] M. Astley and G. A. Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Sixth International Symposium on the Foundations of Software Engineering (FSE-6, SIGSOFT '98)*, November 1998.

[9] C. Callsen and G. Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[10] N. Carriero and D. Gelernter. *How to Write Parallel Programs.* MIT Press, 1990.

[11] P. Ciancarini and C. Hankin, editors. *First International Conference on Coordination Languages and Models (COORDINATION '96)*, number 1061 in LNCS, Berlin, 1996. Springer-Verlag.

[12] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge.* MIT Press, 1995.

[13] J. Ferber and J. Briot. Design of a concurrent language for distributed artificial intelligence. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 755–762. Institute for New Generation Computer Technology, 1988.

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

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

[16] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Intra- and inter-object coordination with messengers. In Ciancarini and Hankin [11].

[17] D. Garlan and D. Le Metayer, editors. *Second International Conference on Coordination Languages and Models (COORDINATION '97)*, number 1282 in LNCS, Berlin, 1997. Springer-Verlag.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[19] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8-3:323–364, June 1977.

[20] S. Jagannathan. Customization of first-class tuple spaces in a higher-order language. In E.H.L. Arts, J. van Leeuwen, and M. Rem, editors, *PARLE '91, volume 2*, number 506 in LNCS. Springer-Verlag, 1991.

[21] S. Jagannathan. Communication-passing style for coordinated languages. In Garlan and Metayer [17], pages 131–149.

[22] JavaSoft. Remote Method Invocation Specification, 1996. Work in progress. http://www.javasoft.com/products/jdk/rmi/.

[23] JavaSoft. JavaSpaces, 1998. Work in progress. http://www.javasoft.com/products/javaspaces/.

[24] G. Kiczales, J. des Riviéres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[25] T. Kielmann. Designing a coordination model for open systems. In Ciancarini and Hankin [11], pages 267–284.

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

[27] W. A. Kornfeld and C. Hewitt. The scientific community metaphor. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1), January 1981.

[28] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison Wesley, 1997.

[29] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1997.

[30] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995. Special Issue on Software Architecture.

[31] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *ACM Conference Proceedings, Object Oriented Programming Languages, Systems and Applications*, pages 276–284, San Diego, CA, 1988.

[32] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of the European Conference on Object-Oriented Programming*, number 512 in LNCS, pages 231–250, 1991.

[33] Object Management Group. CORBA services: Common object services specification version 2. Technical report, Object Management Group, June 1997. http://www.omg.org/corba/.

[34] Open Systems Lab. The Actor Foundry: A Java-based Actor Programming Environment. Work in Progress. http://osl.cs.uiuc.edu/foundry/.

[35] R. Panwar and G. Agha. A methodology for programming scalable architectures. *Journal of Parallel and Distributed Computing*, 22(3):479–487, September 1994.

[36] S. Ren, G. A. Agha, and M. Saito. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing*, 36:4–12, 1996.

[37] H. A. Simon. *The Sciences of the Artificial*, chapter The Architecture of Complexity: Hierarchic Systems. MIT Press, 3rd edition, 1996.

[38] D. C. Sturman and G. Agha. A protocol description language for customizing failure semantics. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*. IEEE Computer Society Press, October 1994.

[39] C. Tomlinson, P. Cannata, G. Meredith, and D. Woelk. The extensible services switch in carnot. *IEEE Parallel and Distributed Technology*, 1(2):16–20, May 1993.

[40] C. Varela and G. Agha. What after Java? From Objects to Actors. *Computer Networks and ISDN Systems: The International J. of Computer Telecommunications and Networking*, 30:573–577, Apr 1998. http://osl.cs.uiuc.edu/Papers/www7/.

[41] J. Waldo. JINI Architecture Overview, 1998. Work in progress. http://www.javasoft.com/products/jini/.

[42] T. Watanabe and A. Yonezawa. An actor-based meta-level architecture for group-wide reflection. In J. W. deBakker, W.P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, number 489 in LNCS, pages 405–425. Springer-Verlag, 1990.

[43] P. Wegner. Coordination as constrained interaction. In Ciancarini and Hankin [11], pages 28–33.

[44] A. Yonezawa, editor. *ABCL An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.