CONCURRENCY—PRACTICE AND EXPERIENCE

Concurrency: Pract. Exper. 2001; **00**:1–48 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

Coordinating Components in Middleware Systems

Matthias Radestock
Trans Enterprise Computer Communications,
The Lee Valley Tecnopark, Ashley Road
London N17 9LN
email rade@tecc.com

Susan Eisenbach

Department of Computing,
Imperial College of Science, Technology and Medicine,
London SW7 2BZ, UK
email sue@doc.ic.ac.uk

SUMMARY

***Configuration* and *coordination* are central issues in the design and implementation of middleware systems and are one of the reasons why building such systems is more difficult and complex than constructing stand-alone sequential programs. Through configuration, the structure of the system is established — which elements it contains, where they are located and how they are interconnected. Coordination is concerned with the interaction of the various components — when an interaction takes place, which parties are involved, what protocols are followed. Its purpose is to coordinate the behaviour of the various components in a way that meets the overall system specification. The open and adaptive nature of middleware systems makes the task of configuration and coordination particularly challenging. We propose a model that can operate in such an environment and enables the dynamic integration and coordination of components through observation of their behaviour.**

KEY WORDS: Coordination; Adaptive Open Distributed Systems; Novel Paradigms

Received

Revised

1. Introduction

We can view a distributed system as a collection of distributed components that interact with each other. The concerns of any distributed system, including middleware systems, can be separated into four parts:

- The *communication part* defines *how* components communicate with each other.
- The *computation part* defines the implementation of the *behaviour* of individual components. It thus determines *what* is being communicated.
- The *configuration part* defines the *interaction structure*, or *configuration.* It states which components exist in the system and which components can communicate with each other, as well as the method of communication. Basically it is a description of *where* information comes from and *where* it is sent to.
- The *coordination part* defines patterns of interaction, ie. it determines *when* certain communications take place.

Inter-part dependencies yield a layered structure (cf. Fig. 1). From a software engineering viewpoint lower layers need not, and should not, know about the higher layers. As far as the lower layers are concerned the upper layers need not even exist. Each of the layers could have its own model, language and implementation (ie. support in a distributed system platform). This clear separation of concerns is extremely beneficial, enabling a high degree of reuse and easier maintenance.

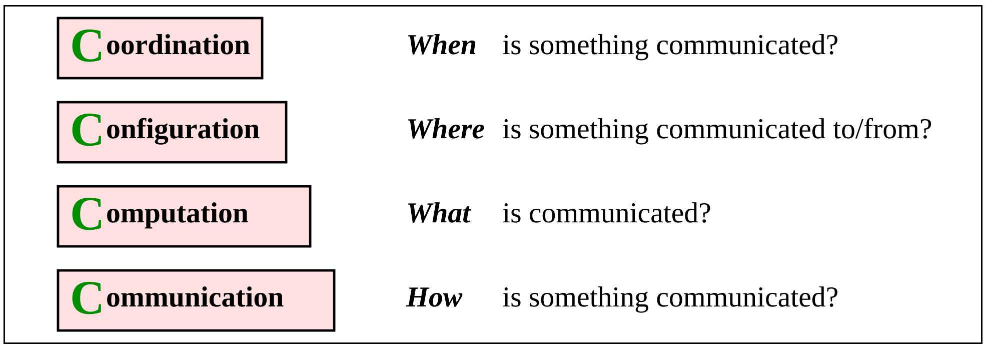
Coordination
When is something communicated?
Configuration
Where is something communicated to/from?
Computation
What is communicated?
Communication
How is something communicated?

Figure 1. The Four Concerns in Distributed Systems

1.1. Dynamic Configuration and Coordination

The interaction structure in middleware systems often changes dynamically: new components are created, existing components are destroyed, connections between components are established and broken up. Such dynamic configuration activities are derived from the functional specification of the system which may state, for instance, that a new member can join a video conference after receiving an invitation. These activities thus need to be triggered by the components in the system themselves, and so the configuration layer needs to be supported by the distributed system platform during the entire life-time of the system in order to enable dynamic access to its functionality.

Coordination specifies patterns of interaction. Such a pattern may, for instance, be that component A can only send message X to component B after component C has sent message Y to component D. Coordination requires configuration — the patterns of interaction need to be specified *before* the parties of interaction; which is precisely the task performed by configuration. We can make a distinction between static and dynamic coordination. In the former case, the interaction patterns are fixed throughout the life-time of a system. In the latter case, interaction patterns are altered dynamically as part of satisfying the application requirements, ie. the changes to the interaction structure and patterns are ultimately triggered by computational components. The coordination layer must exist during the entire life-time of the system. A mechanism is required that enables the interaction with the computation layer.

1.2. Adaptive Systems

A dynamic coordination model allows us to specify systems where all possible dynamic changes to the interaction structure and patterns are known at compile time and are triggered by application components. However, this is insufficient in many large distributed systems, especially middleware systems[27] tasked with enterprise application integration[5]. Such systems are typically long-lived, and require interactive management; both human and automated agents need to be able to reconfigure the system while it is running and even need to be able to alter the specification of the coordination, configuration and computation layers in order to make permanent changes to the overall system behaviour. An example would be a video-conferencing system where some new hardware, say a projection screen, is added to the system during a conference. The components representing the screen need to be added to the system's computation layer and the configuration layer needs to be modified to forward all data of the conferencing communication to that component. Finally we need to alter the coordination layer to ensure that the new component interacts with the rest of the system in the desired manner.

These so-called *adaptive systems* or *evolving systems* are capable of accommodating changes that were not anticipated during the original system development. This is in contrast to static and dynamic systems. Both of these can contain interactive user interfaces or can interact with external components, but such interaction and the resulting changes need to be implemented as *part* of the system functionality; the system functionality itself cannot be altered. Adaptive systems create considerable demands on the capabilities of a middleware architectures and the use of *reflection* [18, 16] as a means of supporting these advanced requirements has been advocated in recent research [25, 6].

1.3. Open Systems

A universal model for configuration and coordination has to be suitable for operating within the context of open systems; it has to be easy to integrate it into existing distributed system platforms and it needs to enable configuration and coordination of existing components without requiring any alterations to them. The model needs to function across heterogeneous systems that may be based on a variety of programming paradigms, languages and platforms. Not only should it be possible to control the configuration and coordination of components in a heterogeneous system, but it must also be possible to control it from the *inside* of the various platforms that make up the system — if configuration and coordination are part of application requirements, then they need to be controllable from potentially any part of the application.

In an open system little is known of the components' implementations and it may even be impossible to alter them. Thus for configuration and coordination to operate in a truly open setting and enable the dynamic integration of components, they must not depend on any knowledge of component behaviour. They certainly should not *rely* on any behaviour specifications, because in general it is impossible to ascertain whether components actually meet them, and hence system safety and security could be compromised. Coordinating components without relying on any explicit behaviour specification is crucial when it comes to middleware systems, where it is important to perform integration with a minimum impact on existing components. Typically, integration is achieved by embedding calls to some special communications API that enable interaction with other system components via the middleware infrastructure. The impact on the existing application in terms of code changes is usually minimal and introducing coordination should not increase this.

1.4. Related Research

The issues of configuration and coordination have received growing attention from the research community, and, as a result, several models, languages and implementations have been proposed and executed. Distributed System standards such as CORBA [20, 22], DCE [28] and RM-ODP [7], and their implementations, address the issue of configuration by introducing a *brokering* mechanism which matches requests by components for particular services with components providing these services. With this basic building block in place, most configuration issues can be addressed. However, coordination is not addressed at all and left entirely to the programmer of the components.

Formalisms, such as Gamma [2] and languages such as Linda [14, 3] have emerged. However, they are not aimed at integration with existing systems or operation in an open environment. Furthermore, only limited facilities exist for re-using coordination patterns, and coordination is typically embedded in application code rather than being separated. Research in software architecture [13, 21, 12, 26], by contrast, has placed considerable emphasis on layer separation. However, the distinct role of coordination has only been recognised recently. Consequently several systems have emerged that address coordination issues, usually as extensions to existing systems. Examples of this are *ToolBus* [4] (an extension to the *PolyLith* software bus [24]), ConCoord [15] and Midas [23] are extension of Darwin [17]. ActorSpace [8] is an extension of an actor language and *Manifold* [1] is based on a model where processes communicate anonymously via streams. Common to all approaches is the lack of openness — coordination in these systems relies on particular features that are unique to the specific system. Dynamic integration of existing components is usually possible, but only for components that have been designed, implemented and compiled for the particular system used. Dynamic change is supported, but systems cannot adapt to changes in the requirements that go beyond the scope of the original specification. Furthermore, the above coordination mechanisms only provide limited means of abstraction, ie. the construction of patterns of coordination and their reuse. This is mainly due to the use of separate coordination languages that lack expressiveness.

1.5. Outline of Our Approach

Our aim is to enable coordination in adaptive and open distributed systems, such as middleware systems. Further to that, we want to be able to integrate components on the level of source code, object code and running code, including existing and running legacy applications. The key element in our solution is a mechanism that enables the observation and coercion of dynamic component behaviour through the interception of messages. The first part of this paper is devoted to the description of this so-called *traps* model. Traps employ a sophisticated type system for specifying message patterns and rules for defining actions to be taken when messages have been intercepted. The patterns and rules can be altered dynamically and thus traps represent a dynamic configuration and coordination layer. Since traps operate without having any knowledge of the behaviour of the components, they do not depend on any component interface/behaviour specification. Traps integrate the configuration and coordination layers into the computation layer without jeopardising the benefits of clear separation. Thus coordination can be designed and implemented using the same techniques deployed in the design and implementation of the application components. As a result, coordination code can be reused in the same way as application code. The approach also enables *meta coordination*; the coordination of coordination itself. In the final part of this paper we use the well-known example of the Dining Philosophers to illustrate how our model can deal with various, increasingly complex, coordination tasks.

2. Traps — A New Model For Coordination

In order to facilitate configuration and coordination in an evolving heterogeneous distributed environment, we need to devise a suitable model that has very few demands on the system architecture and is thus easily incorporated into both existing and new systems. The first step in devising our coordination model is to take a slightly different view of the message-based communication model. This new view is illustrated in Figure 2. When a component A sends a message to another component B, the message gets stored in a location of the so-called *message space*, based on its *type*. From that location it is then forwarded to the receiving

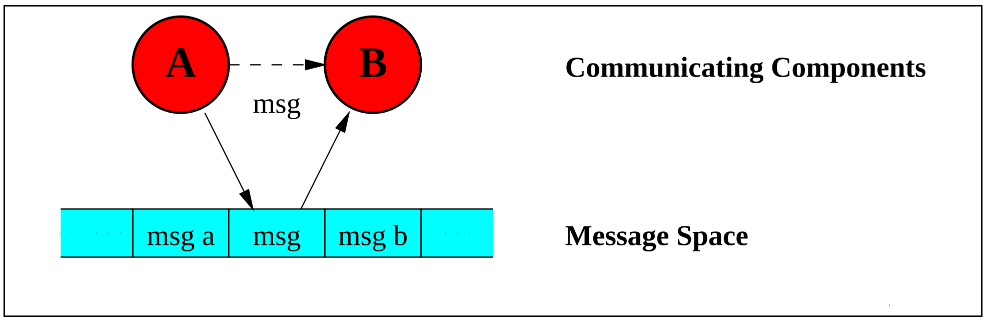

Figure 2. Communication via Message Spaces

component. It should be noted that this transformation of our view of the communication model happens on the *conceptual* level, unlike, for instance, in Linda where the tuple-space model is exposed to the programmer. The new view is transparent to the components involved; as far as component A is concerned it is still sending a message to component B, and as far as component B is concerned it is still receiving a message from component A. Conceptually though we can view things differently. Component B is *notified* of an ‘interesting’ activity: a message that component A is trying to send to component B.

2.1. Message Types

A message between two components consists of

- the *originator*, ie. the component that sent the message,
- the *recipient*, ie. the component that is the intended recipient of the message,
- the *content*, ie. the data elements, and

- the *context*, ie. additional information required by the communication and coordination layers, such as time stamps and request ids.

The *type* of a message encompasses those elements that are visible to the programmer, ie. everything apart from the context information. It can therefore be defined as

$$\mathsf{Message} = \mathsf{Component} \times \mathsf{Component} \times \mathsf{Component}^*$$

That is, the product type of components (originator), components (recipients) and sequences of components (message content). Locations in a message space correspond to message types, hence messages sharing the same originator, recipient and content are stored in the same location. Some examples of message types (in pseudo-code) are:

`device=>handler()`	an empty message from *device* to *handler*
`device=>handler(handle,data,12)`	a message from *device* to *handler* with three components as content: *handle, data* and *12*.

We do not attach any special significance to the first element of the message content. In many object-oriented systems this will be the name of a method to be invoked, however, our model operates on a more abstract level and can therefore be oblivious to this special semantics.*

*The `=>` in our notation should not be confused with the `->` method invocation construct found in languages like *C++*. In our notation the element to the left of the arrow is the *sender*, the element to the right is the *recipient* and the arguments follow.

2.2. Message Patterns

A *message pattern* defines a subset of the domain of message types. Its domain can therefore be defined as the power-set of message types, ie.

$$\mathsf{MessagePattern} = \mathrm{P}(\mathsf{MessageType})$$

Message patterns are used by the programmer to identify interesting messages, messages requiring special treatment by the coordination layer. They typically use the type system of the underlying programming language. However, it should be noted that the type system ought to be sophisticated, with the ability to dynamically construct types from instances and not just other types. If these capabilities are not present then a separate type system must be introduced to complement the existing one. Examples of some more sophisticated message patterns are:

```
device=>handler(handle,data,12)

Device=>Handler,'special(handle,Any)+^String

Device=>handler()+Any,Device=>Device(transfer)

Any=>Any()+Any
```

The first pattern covers exactly one message. The second pattern covers all messages from components of type *Device* to components of type *Handler* or the symbolic component *special*, with at least two arguments, the first of which must be the component *handle*, the second of which can be of any type, and the remaining arguments being of a type other than *String.*[†] The

[†]Upper case identifiers in our pseudo-code denote *types*, lower case identifiers denote *variables* holding component references and identifiers prefixed with a single quote denote symbols.

third pattern covers all messages from components of type *Device* to the component *handler* with any number of arguments of any type, and messages between components of type *Device* with *transfer* as an argument. The fourth pattern covers all messages.

As can be seen from these examples, a sophisticated type system enables the concise specification of very complex patterns. Traps do not inherently depend on such type systems though, as there are other places in the trap system where such complex decisions can be made. However, the more expressive the type system is, the less computationally expensive the introduction of traps becomes.

Having introduced the notion of message types, we now view communication between two components in the following way: Messages, instead of being sent directly to their intended recipients, are stored in locations of the so-called *message space*. Locations in a message space correspond to message types, hence messages sharing the same originator, recipient and content are stored in the same location. From that location they are then forwarded to the receiving component. It should be noted that this transformation of our view of the communication model happens on the *conceptual* level, unlike, for instance, in Linda where the tuple-space model is exposed to the programmer. The new view is transparent to the components involved; as far as the sending components are concerned they are still sending messages in the usual way and as far as the receiving components are concerned they are still receiving messages in the usual way. Conceptually though we can view things differently; the receiving components are *notified* of an 'interesting' activity: a message that some component is trying to send to it.

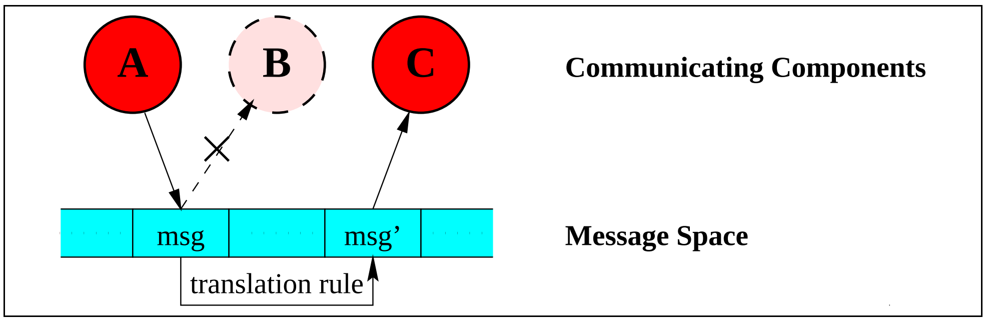

Figure 3. Message Relocation by Translation Rules

2.3. Translation Rules

Our message-based communication model is obviously of not much use if all that happens is essentially the same as before; locations in the message space just serve as 'trampolines' that bounce messages to their target components. What we require for coordination is some means of altering the flow of messages. We achieve this by installing *translation rules* at locations in the message space. These translate the messages at the location into other messages, thus relocating them to different places in the message space and effectively intercepting the message.

The translation rule in Fig. 3 translates the original message (from component A to component B) into a message that has component C as the target. The new message could be seen as having precisely the meaning presented earlier as an alternative perspective, ie. component C is notified of the attempt by component A to send a particular message to component B. Component C thus conceptually resides in the coordination layer. It could coordinate all the activities between component A and B if translation rules were specified that relocate any messages exchanged between the two components. Thereby C could act as a simple forwarder, or could accomplish arbitrarily complex coordination tasks, such as protocol translation and enforcement, interaction with other coordinators etc.

In our model, translation rules always translate messages into new messages where the originator is the message wrapper of the original message, ie. an encapsulating component for the original message.[‡] Further, the recipient and content of the new message does not depend on the original message. A translation rule thus simply specifies a new recipient and content:

$$\mathsf{TranslationRule} = \mathsf{Component} \times \mathsf{Component}^*$$

Such translation rules can be defined completely independently of the underlying programming language since they do not perform any computation whatsoever. This keeps the semantics simple, offers opportunities for easy and efficient implementation, and enables deployment in a heterogeneous language/platform setting.

When a translation rule is applied to a message, the resulting message contains the message wrapper of the original message as the originator. The recipient and content are supplied by the translation rule

```
logger('io-event)
```
[§]

applied to a message

```
device=>handler(handle,data,12)
```

[‡]The purpose of message wrappers is to expose messages as components in the programming language, even though messages themselves may not be components.

[§]In our pseudo-code we define translation rules in the same way as message types, except that the originator and following `=>` are not present.

will result in a message

```
device=>logger('io-event,[device=>handler(handle,data,12)])
```

where the square brackets denote the message wrapper of the original message.

Messages resulting from the application of translation rules get stored at their appropriate locations in the message space. Hence they can be subject to further translation. Eventually the messages cannot be translated any further and are sent to their intended recipient. Since translation rules always generate messages containing an encapsulation of the original message as part of the content, the elements of the original message, such as the original recipient, can all be used in the further decision process by coordination components.

2.4. Defining Traps

Placing a translation rule on a location in the message space is the equivalent of 'setting a trap', hence the name of this model. Instead of being bounced back and delivered to the intended recipient, a trapped message undergoes translation. The same translation rule often applies to many locations in the message space. As we noted before, the number of locations in the message space can be very large or even infinite. It is therefore impossible to install translation rules individually at every location in the message space. Hence a trap definition consists of two components:

- a *message pattern*— using the described type system for messages, this defines a subset of the domain for messages, ie. locations in the message space. Messages in the subset are caught by the trap.

- a *translation rule*— using a new recipient and message prefix, this translates it into a new message.

Thus, traps can be characterised as

$$\begin{aligned}\mathsf{Trap} &= \mathsf{MessagePattern} \times \mathsf{TranslationRule} \\ &= \mathsf{MessagePattern} \times \mathsf{Component} \times \mathsf{Component}^*\end{aligned}$$

In our pseudo-code we define traps using a `>>` operator. For instance, the trap

```
Device=>Handler(handle)+Any >> logger('io-event)
```

will trap all messages sent from devices to handlers with *handle* as the first argument plus any number of further arguments of any type. It will translate these messages to messages to the component *logger*, with the symbolic component *'io-event* as the first argument and the encapsulated original message as the originator. Note that message patterns are part of the type system and message wrappers can be matched against them. This enables the specification of traps that further translate a message that has already undergone some translation. For instance, messages generated by the above trap would match the pattern

```
[Device=>Handler(handle)+Any] => logger('io-event).
```

2.5. Matching Policies

When a message is matched against the message patterns of the currently installed traps, it is possible that it matches more than one pattern. In dealing with this situation, we have a choice between two matching policies:

1. Message translation is performed by all traps whose message pattern matches a message.
2. Message translation is performed by the traps whose message pattern matches the message most specifically, compared to the other patterns.

Both policies are useful in certain contexts. The first policy would be employed in cases where several independent coordinators are interested in a message and therefore install traps to intercept it. For instance the two traps

```
Device=>Handler()+Any >> forwarder('io-event)
Device=>Handler(handle)+Any >> logger('io-event)
```

could be installed completely independently; one in order to forward messages, one in order to log a subset of the messages. We would actually want both coordinators (ie. the `forwarder` and `logger`) to deal with messages matching both patterns, instead of a selection being performed based on the most specific message pattern (which in the above case would select the second trap in preference to the second). The second policy is typically employed in cases where a single coordinator installs several traps; more general traps for dealing with 'normal' messages and specific traps for dealing with 'exceptional' messages requiring special coordination, e.g.

```
Device=>Handler()+Any >> forwarder('io-event)
Device=>Handler(handle)+Any >> forwarder('handle-io-event)
```

In order to deal with these two cases we therefore implement the following policy:

> Traps with the same new recipient form a *trap group*. When a message matches the message patterns of several traps in the group, then only the translation rule of the trap with the most specific matching message pattern is invoked.

Message translation is performed by all trap groups that contain traps with patterns matching the message.

Trap groups define the boundaries of pattern-based selection and ensure that a message is not translated into two messages with the same recipient. Thus, if all the above traps were installed, a message

```
device=>handler(handle,data,12)
```

would be translated into two messages,

```
[device=>handler(handle,data,12)]=>forwarder('handle-io-event)
[device=>handler(handle,data,12)]=>logger('io-event).
```

which is exactly what we would expect.

There is a special case involving the pattern-based selection — when several patterns match a message but neither of them is more specific than any of the others. In the simplest case this will occur when two patterns are identical. The policy we employ in this case is to select the most recently installed trap, thus ensuring a deterministic outcome of the selection process.

2.6. Coordination Protocol

When integrating a component into a system, the system needs to be configured in a way that translates messages sent from the component to other components into messages the other components understand and visa versa. This is accomplished by enclosing the component in a container that performs the required translation, and by defining traps that intercept messages sent from the component. Traps are applied recursively in order to translate a messages, resulting in a set of messages that logically replaces the original message. Every message in the message set is delivered to its new recipient. If the original message was a request requiring a reply then the first reply to any of the messages in the message set is sent back to the originator of the original message as the reply to the request. All further replies to messages in the message set are ignored, thus ensuring that at most one reply is returned. Typically all intercepted messages are translated into messages to the container component which in that case contains logic to perform the actual message translation. Thus the container component becomes a true wrapper around the integrated component, dealing with both the incoming and outgoing messages.

Configuration with traps is thus a function that is performed just after a message has been *sent*. By contrast, coordination is performed just before a message is *delivered*. It is concerned with ensuring that the processing of messages of an integrated component is synchronised, depending on the application logic, with the processing of messages by other components in the system. In our model, coordination is accomplished by encoding the required logic in components that receive intercepted messages. The principal decision to be made by the components is when the message should be submitted for processing to the original recipient. In order to make that decision, the coordination components need to interact with each other. This causes a software engineering problem because the coordination logic is often a composite entity whose elements are unaware of each other and hence cannot engage in any explicit interaction. This composite nature of the coordination logic is a result of the composite nature of applications — they are built out of components which each have their own coordination logic and are 'glued together' by yet more coordination logic. To overcome this problem we

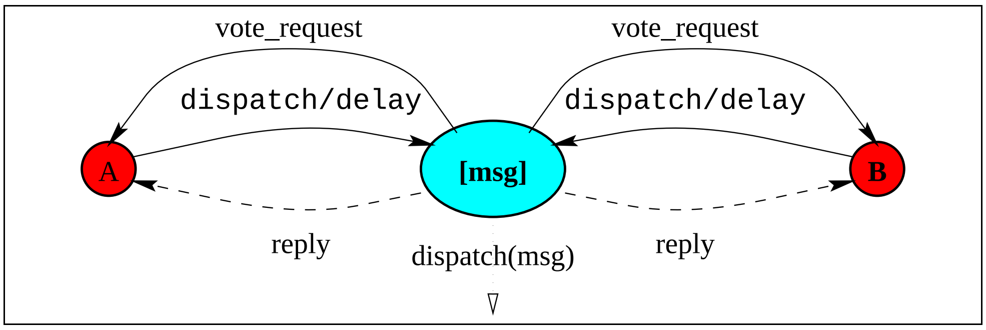

Figure 4. Coordination Protocol

add a *coordination protocol* to our model that manages the interaction between coordinators transparently, ie. in a way that doesn't require coordinators to know about each other. As a result, coordinators can be implemented as state machines that perform state transition whenever an interesting event occurs and whose states corresponds to sets of constraints on occurrences of events.

When a message is intercepted by traps, it is eventually translated into a set of messages that cannot be further translated. Then the following happens (cf. Fig. 4):

1. The message wrapper of the original message initiates a *round of voting*. Messages in the message set are interpreted as *requests for votes* and dispatched.
2. The message wrapper waits until the same number of votes as requested, have been received. Participants in a vote submit their vote through sending a `dispatch` or `delay` message. They expect to receive a reply to that message, containing the outcome of the round of voting.

3. If all participants voted for `dispatch` then the outcome is `dispatch`. The original message is dispatched, ie. sent on its way to its destination.
4. If one participant voted for `delay` then the outcome is `delay`. Nothing is done.
5. The reply messages with the vote result are sent to the participants.

The complexities of the coordination protocol can be hidden from the programmer by splitting coordination components into two separate components; a *protocol wrapper* and a *logic wrapper*. Programmers need only to be concerned with the logic wrapper which, typically, implements some kind of state machine. All that the component has to do is vote on a message and perform a state transition if the vote succeeds. The protocol wrapper, which can be automatically created by the system, controls the message flow to and from the logic wrapper and implements the coordination protocol. It takes care of message ordering and re-voting and isolates the programmer from any changes that may be made to the protocol over time.

An important property of our coordination protocol is that it ensures *fairness in the absence of progress*; a message that can be processed will be processed no later than at the time were no other message is being processed by any component in the system. This ensures that applications do not come to a standstill because messages whose processing is required to enable the processing of other messages, are being delayed indefinitely despite being enabled by the applications coordination logic.

3. An Example

We shall now demonstrate how configuration and coordination in applications can be accomplished with traps, using the well-known example of the 'Dining Philosophers'. Philosophers sit around a table with food. There is a chopstick between every two philosophers. Philosophers require both their left and right chopstick in order to eat. A chopstick can only be held by one philosopher at a time. It needs to be ensured that philosophers don't starve — we need to prevent situations of deadlock and livelock and ensure fairness. Philosophers and chopsticks are to be treated as 'given' types of components, ie. we do not have access to their source code and hence cannot modify it. Neither do we have any detailed knowledge of the components' behaviour. Thus coordinating philosophers and chopsticks in the context of Dining Philosophers is very similar to the integration/reuse of 'legacy' components in a heterogeneous distributed system, such as a middleware system. We investigate two variations of the example, with increasing degree of complexity and show how the additional complexity is primarily accommodated in an incremental fashion without the need for rewriting existing code.

3.1. Configuration

Initially, we deal with static configuration only, ie. we create a certain number of philosophers and chopsticks and establish a configuration where philosophers are assigned chopsticks in the manner specified. We are assuming that coordination is performed by the philosophers and chopsticks themselves, which therefore have to be aware of the context they are being used in. When philosophers and chopsticks are created, the required coordinators are set up as well.

After all coordination logic is in place, the philosophers are told to start eating. A philosopher attempts to pick up a chopstick by sending a `pick` message to the symbolic components `'left` or `'right`. It drops a chopstick by sending a `drop` message. We assign a coordinator to each philosopher, which installs traps that intercept the `pick` and `drop` messages of a philosopher and translate them into messages to the coordinator. Upon receipt of one of these messages the coordinator replaces the original `pick` or `drop` message with a `get` or `put` message to the actual left or right chopstick.

3.2. Coordination

To facilitate reuse, philosophers and chopsticks should be completely unaware of the context in which they are being used, specifically philosophers should not require knowledge that picking up a chopstick requires coordination with other philosophers. Philosophers should be able to attempt picking up a chopstick at any time and dropping a previously picked up chopstick at any time. They can attempt to pick up both chopsticks at the same time or one after the other or pick one up and then drop it again. The only assumptions we make, for simplicity's sake, is that philosophers will not pick up a chopstick they currently hold, and will not drop a chopstick they do not currently hold. The result of this liberal approach is that a great variety of component implementations can fulfil the roles of philosophers or chopsticks. Thus there is a wide scope for reusing existing components in that role without the need for modification.

We coordinate the dining philosophers by ensuring that when a philosopher attempts to pick up the first chopstick, the request is delayed until *both* chopsticks are available. When the first chopstick is being picked up by a philosopher all requests by other philosophers to pick up

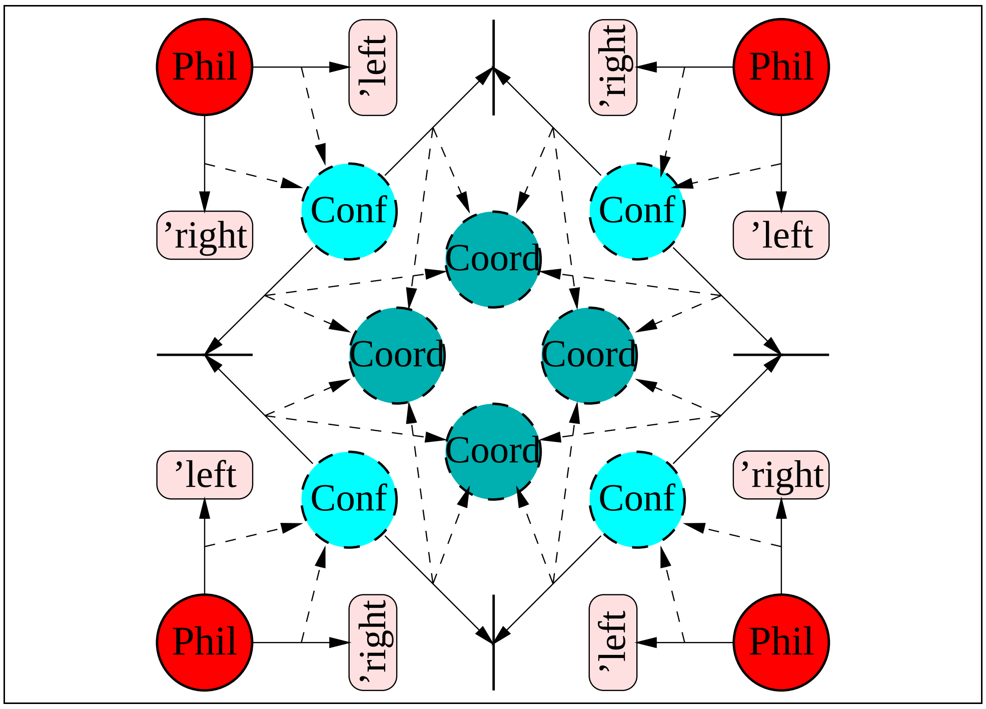

Figure 5. Coordinating the Dining Philosophers

the complementary chopstick are delayed. This policy ensures both freedom of deadlock and livelock as well as guaranteeing that chopsticks are only picked up by one philosopher at a time. In order to implement the coordination policy, coordinators create an auxiliary component. Traps are installed that intercept messages to the philosopher's chopsticks (cf. Figure 5). One set of traps intercepts the messages sent by the philosopher's coordinator, another set has a message pattern which matches messages to the chopsticks from *any* component. Since the latter pattern is less specific than the one of the first set of traps, these traps will intercept all messages sent to the chopsticks by other coordinators. The intercepted messages are submitted for synchronisation to the auxiliary components. The auxiliary components implement a state machine encapsulating the coordination policy by delaying `get` messages.

3.3. Dynamic Change

In the previous two variations of the Dining Philosophers, the configuration of the system remained unchanged once it had been set up. However, coordination is often required in settings where components are created dynamically. We shall now demonstrate how this can be accomplished in a variation of the Dining Philosophers example in which we allow new philosophers to join the existing ones.

New philosophers are placed at the ‘end’ of the table, ie. next to the last philosopher. A new chopstick is created that will be shared between the new philosopher and the last philosopher. The coordinator of the last philosopher needs to be notified of the changed configuration so that it can amend its existing traps. We do this by sending it a `changeL` or `changeR` message, depending on whether the new philosopher is seated to the left or right of the last philosopher. The message contains a reference to the new chopstick and causes all existing traps of the coordinator to be removed and new traps to be installed. The resulting changes in the system configuration are illustrated in Figure 6.

Replacing a chopstick is not always safe. We cannot replace it while it is held by the philosopher in question. We therefore amend the coordinator state machine to defer the `changeL` and `changeR` message in certain states. Furthermore, it is important to initialise the coordinator of the philosopher in a way that reflects the current state, ie. which of its two chopsticks are currently held by other philosophers. We accomplish this by getting the

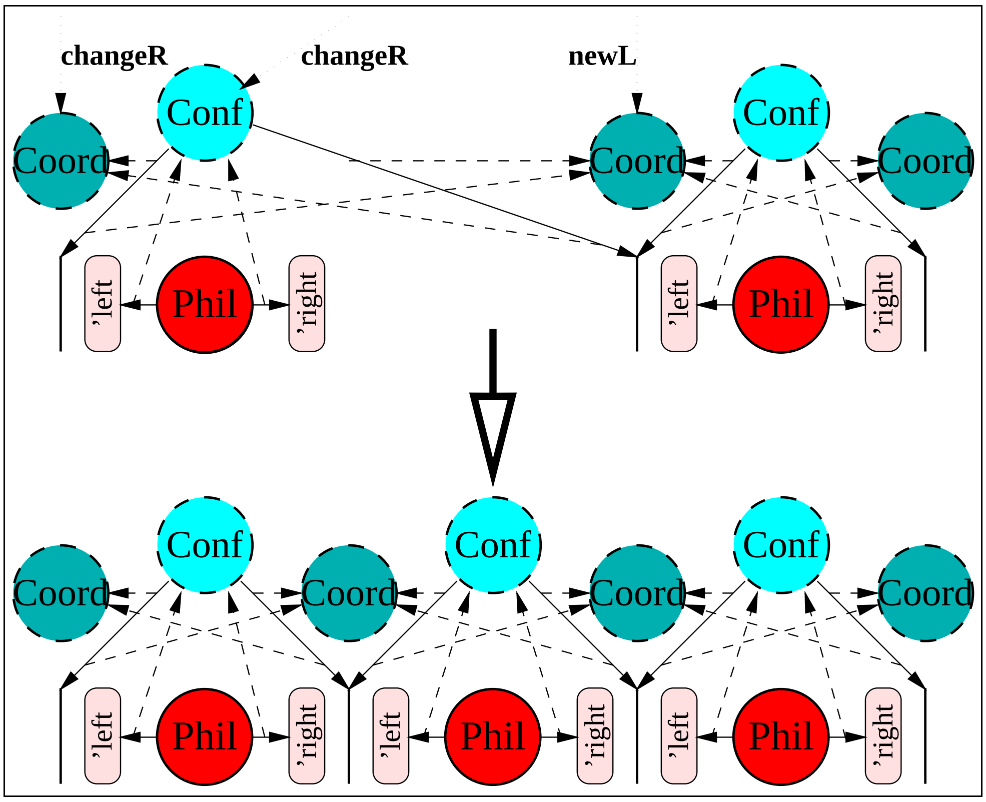

Figure 6. Introducing a new Philosopher

initialisation code of the coordinator to ask the chopsticks whether they are currently being held by anyone. The traps for the coordinator are installed *prior* to submitting these requests, thus ensuring that the results obtained are current and won't change without involvement of the coordinator.

4. Performance Evaluation

Our model introduces overheads that affects performance in two ways:

1. The "fixed" cost of coordination, i.e. what overhead is incurred by introducing our coordination model into a system without actually using it.
2. The "variable" cost of coordination, i.e. how coordination affects scalability — the ability of a system to cope with an increased workload by increasing resources, e.g. processor speed, memory, number of nodes, network bandwidth.

Below we investigate each issue in turn.

4.1. Test Framework

Our main interest is in systems where coordination primarily takes place between components executing in a single address space in a multi-threaded environment. Hence we assess the fixed and variable cost of coordination in our model (and its implementation in TECCware [29]) by measuring the average delay to the processing of messages in a generic framework that permits the simulation of various coordination scenarios in precisely such an environment. However, we will also discuss the implications of our findings for the more general case. Note further that we focus on coordination, but the results are equally applicable to configuration since it is based on the same algorithms as coordination.

Figure 7 shows our generic framework. A stream of messages is fed to each of $n_{work} = n$ worker components $W1 \dots Wn$. There are $n_{coord} = m$ coordinator components $C1 \dots Cm$. Every coordinator coordinates every worker. This is a worst-case scenario since in most cases

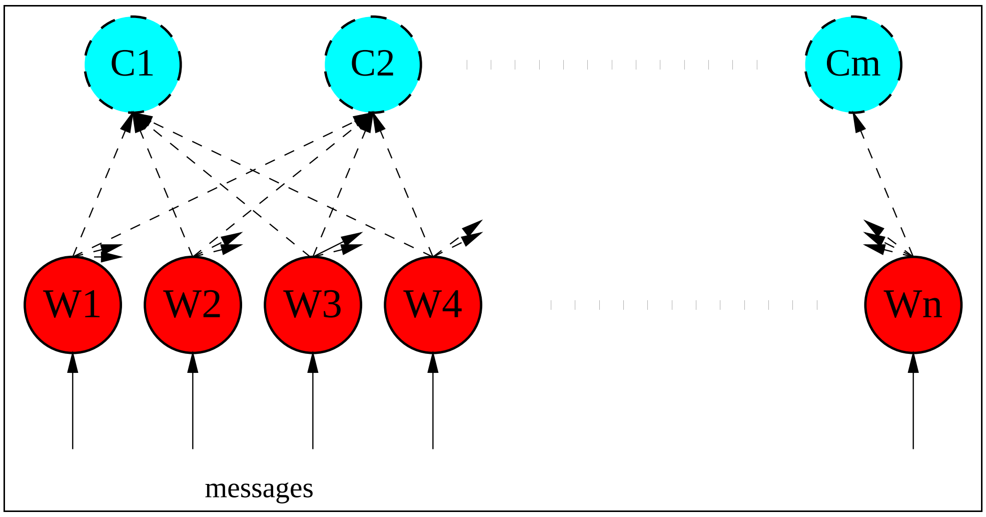

Figure 7. Performance Test Framework

coordination can be accomplished by coordinating smaller groups of workers. For instance only two chopsticks are coordinated by every coordinator in our Dining Philosopher example, regardless of the total number of chopsticks. Note that we are interested in measuring overheads, and hence both the worker and coordinator components in our test framework do not actually perform any work upon receiving a message.

4.1.1. Measurements

The framework allows us to measure the impact of the following factors on the average message delay:

Workload. An increase of workload, i.e. the number of messages needing processing within a certain period of time, can be dealt with architecturally by increasing the number of workers n_{work}. For this to result in any improvement, the number of processors (and processes) and/or the number of nodes should be increased as well, i.e. a new worker should be placed on a new processor/node. We perform our tests on a single single-processor machine and hence we can measure the *overhead* that is associated with an increased number of workers. The size of this overhead, and how it increases with the number of workers and in different coordination settings can limit scalability since it might cancel out any gain obtained from the availability of additional resources.

Degree of Coordination. By varying n_{coord}, the number of coordinators, we can measure the impact of the *degree of coordination* — i.e. the more coordinators are interested in an intercepted message the "more" coordination takes place.

Probability of Failure. By altering the constraints enforced by coordinators, we can control $p_{disabled}$, the probability of an intercepted message not being enabled for processing. Changes in the value reflect different "styles" of coordination. For instance, an application programmer could use coordination in order to delay the processing of messages in certain exceptional circumstances (and hence $p_{disabled}$ would be very small) or in order to control access to a heavily used resource (in which case $p_{disabled}$ could be very large).

Our aim is to ascertain how the degree of coordination and probability of failure affect scalability, i.e. how particular settings of n_{coord} and $p_{enabled}$ affect the message delay at increasing workloads generated by an increase in the number of workers n_{work}.

4.1.2. Optimisations

In our implementation we chose not to dispatch vote requests in parallel because there is nothing to be gained from doing so in our setting:

- Parallel voting reduces communication overheads, but since we are operating in a single address space, these are very low anyway.
- Parallel voting allows parallelism between coordinators. However, this would only result in a performance gain if the coordinators each had their own resources, i.e. a node/processor. This is not the case in the kind of systems we are interested in, where system resources would typically be allocated to workers rather than coordinators.

Performing the voting sequentially, allows us to perform an optimisation: we establish an ordering on coordinators and always dispatch vote request in the same order. This practise is widely used in databases for prevention of deadlock during the acquisition of table locks. In our test framework (and indeed in most other coordination scenarios) this means that any potential deadlocks are detected by the first coordinator and no vote requests are sent to any of the other coordinators in such a case. Even in a fully distributed setting, where each coordinator has its own resources, sequential voting will usually be more efficient since the optimisation can significantly reduce the amount of "unnecessary" work performed by coordination controllers.

The optimisation enables us to refine the formula which determines the average message delay as follows:

$$\begin{aligned} t_{message} &= (n_{abort} + n_{fail} + 1)(t_{constraint} + n_{traps} t_{match}) + \\ &\quad (n_{abort} c_{abort} + n_{fail} c_{fail} + n_{coord})(t_{constraint} + t_{roundtrip} + t'_{vote}) + \\ &\quad n_{coord}(t_{roundtrip} + t'_{ack}) \end{aligned}$$

where n_{abort} is the average number of times a vote is aborted, c_{abort} is the average number of coordinators that vote before a vote is aborted, n_{fail} is the average number of times a vote fails due to constraints not being satisfied, and c_{fail} is the average number of coordinators that vote before a vote fails.

In our test scenario c_{abort} is 1 because all workers share the same set of coordinators and hence the first coordinator in the established coordinator order will perform any necessary aborts. Furthermore, all coordinators enforce the same constraints and hence c_{fail} is 1. Consequently the above formula can be simplified as follows:

$$\begin{aligned} t'_{message} &= (n_{abort} + n_{fail} + 1)(t_{constraint} + n_{traps} t_{match}) + \\ &\quad (n_{abort} + n_{fail} + n_{coord})(t_{constraint} + t_{roundtrip} + t'_{vote}) + \\ &\quad n_{coord}(t_{roundtrip} + t'_{ack}) \end{aligned}$$

4.2. Constant Overheads

4.2.1. Ordinary Message Delivery

The vast majority of messages in any application will typically be between uncoordinated components since coordination tends to take place on a coarse-grain level and the finer-grain components, which perform the bulk of the work, remain uncoordinated. Our model does not add any overheads to these very common interactions. There is, however, an inherent cost of interaction in any system. By measuring it we can later compare it with coordination overheads.

By setting n_{coords} to 0 we ensure that no traps are installed on worker components. This means that message delivery to the components remains unaffected by coordination. Since our workers do not actually perform any work, the average time it takes to process the message is equivalent to the average dispatching overhead. The value we obtain by performing these measurements on a dedicated Sun Sparc Ultra/300 workstation is 0.054ms.

4.2.2. Reflection

The implementation of our model uses reflection, specifically reflected mailboxes. By measuring the overhead associated with the use of reflection we can determine a "base line" — messages that are subject to coordination in our implementation cannot possibly be subject to a smaller overhead than that of reflection. In order to determine this base line we carry out tests in the same system as above except that worker components use enabled sets and hence have reflected mailboxes. The average per-message overhead we measured in this setting is 0.13msg. The results show that the cost of mailbox reflection is significant; it doubles the message delivery overhead.

4.2.3. Coordination Controllers

The two scenarios from above identify the overhead associated with sending messages to ordinary components and to components with reflected mailboxes. Both of these are not affected by our coordination model. These are the vast majority of messages in an application. The third most common category are messages to coordinated components that are themselves not subject to coordination, i.e. no traps have been installed for them. We can determine the overhead associated with the delivery of such messages by setting n_{coords} to 1 and letting the coordinator install non-matching traps on the workers, i.e. traps that do not match the messages received by them. This forces the workers' mailboxes to become reflected and replaced by trap-aware mailboxes which serve as coordination controllers. The overhead we measured in this setting is 1.28ms.

The result shows that the implementation of coordination controllers as reflected mailboxes creates a very significant overhead — approximately ten times that of basic reflection and twenty times that of ordinary message delivery. The fact that the overhead is larger than that of basic reflection is not surprising since a message received by a coordination controller has to undergo more checks before being scheduled for processing than in the case of ordinary reflected mailboxes. However, this does not really explain the size of the overhead, i.e. the fact that it is *ten times* that of basic reflection. The reasons for this considerable overhead are inherent inefficiencies in our implementation — it is a *prototype* and emphasis was placed on the *clarity* of the implementation rather than efficiency. For the same reasons no optimisations were performed for dealing with uncoordinated messages and hence a lot of unnecessary work is performed when dealing with such messages.

4.2.4. Pattern Matching and Constraint Evaluation

We are also interested in determining the impact of pattern matching and constraint evaluation — every received message needs to be matched against local constraints and the message patterns of potentially all traps installed on the worker. We measure the impact of the number of traps on the per-message overhead by getting the coordinator to install 128 traps instead of just 1. We obtain a result of 4.76ms.

The result implies that the overhead per trap is just under 0.03ms. Since typically only very few traps are installed, the contribution of this overhead to the overall per-message overhead is negligible. Constraint matching uses the same algorithms as pattern matching; they are both based on type comparison. As with traps, the number of installed constraints are typically be very small and hence the contribution of the overhead to the overall per-message overhead is negligible.

4.3. Variable Overheads

4.3.1. Workload

Figure 8 illustrates the increase in the overhead that results from an increase in the number of workers. We can see that the rate of increase seems to be rather low initially and eventually converges upon a value close to 100 percent, i.e. doubling the number of workers doubles the overhead.[¶]

[¶]Note that both the x and y axis are scaled logarithmically.

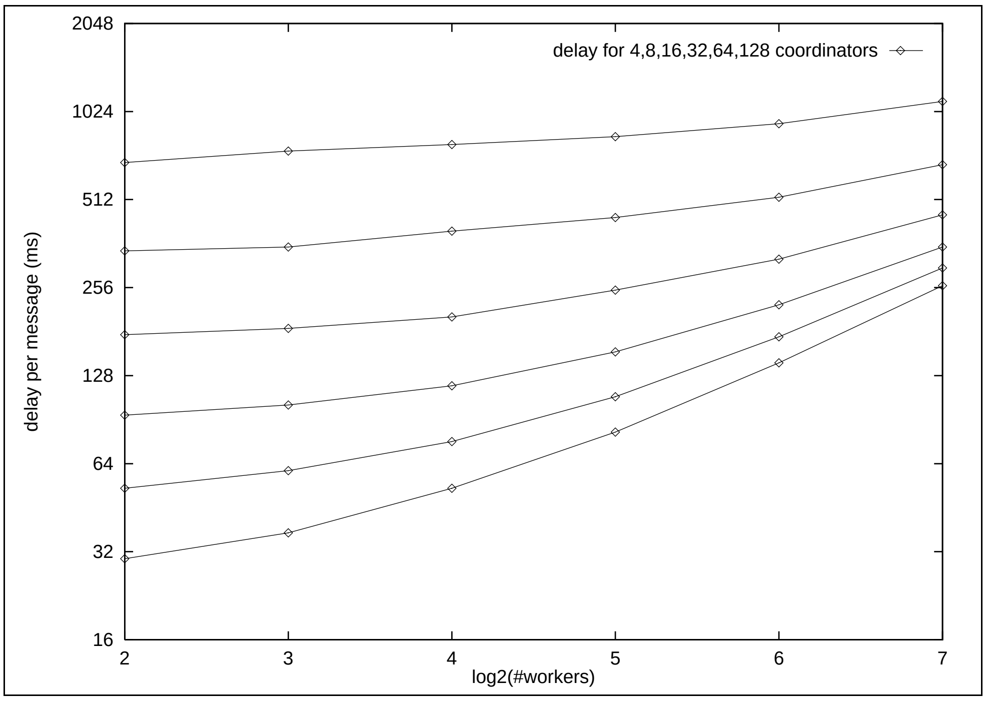

Figure 8. Dependency of the overhead on the workload

The result can be explained as follows: In the worst case scenario a coordinator is asked simultaneously by each worker to vote on a message. This may result in $n_{work} - 1$ polls being aborted and one succeeding. Messages are aged artificially when polls have been aborted in order to prevent repeated aborts, but it may take up to n_{work} (and an average of $n_{work}/2$) attempts before a message is considered successfully, because by then the message will have reached age n_{work} and necessarily be the oldest. Thus, doubling n_{work} doubles n_{abort}, the average number of times a message is being aborted, which in turn doubles the overhead. For large numbers of coordinators the overhead is masked by the overhead associated with the

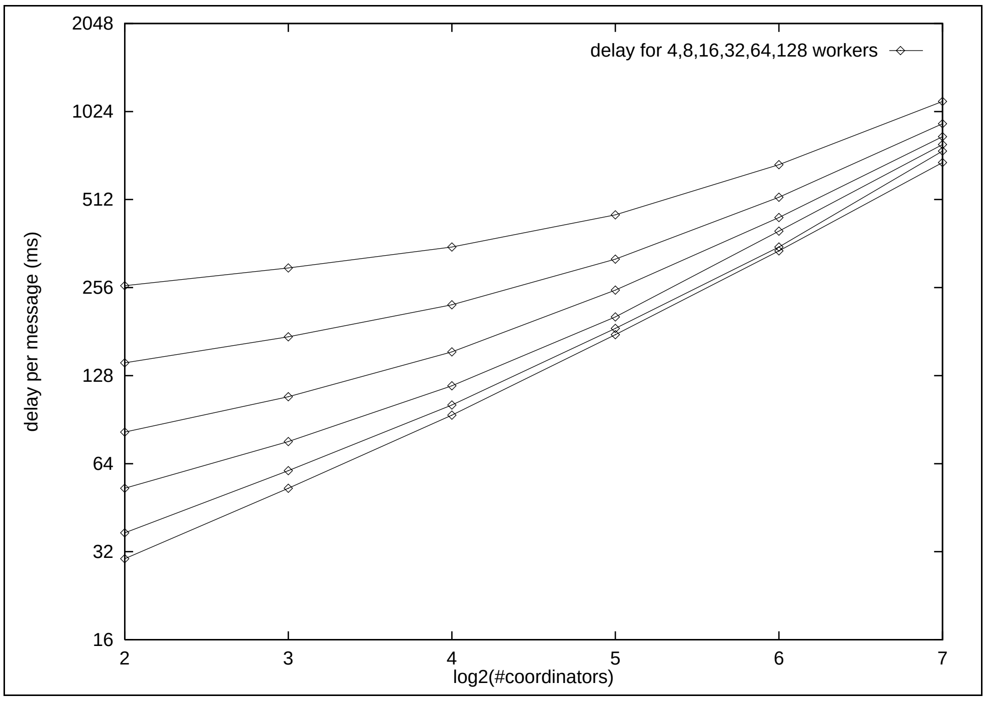

Figure 9. Dependency of the overhead on the degree of coordination

degree of coordination (see below) and hence the 100 percent rate of increase will only be seen with a much larger number of workers, e.g. the diagram shows that the 100 percent rate of increase is reached at 2^6 workers for 4 coordinators and at 2^7 workers for 8 coordinators.

A distributed setting will not improve this result since most of the overhead occurs at the coordinators and their resources are not increased by placing new workers at new nodes. Furthermore, the communication overhead is increased.

4.3.2. Degree of Coordination

Figure 9 illustrates the increase in the overhead that results from an increase in the number of coordinators. As in the above scenario, the rate of increase seems to be rather low initially and eventually converges upon a value close to 100 percent, i.e. doubling the number of coordinators doubles the overhead.

The result can be explained as follows: The consideration of a message requires interaction with all coordinators. An overhead is associated with each of these interactions and doubling the number of coordinators doubles the overhead. For large numbers of workers this overhead is masked by the overhead associated with the workload (see above) and hence the 100 percent rate of increase will only be seen with a much larger number of coordinators, e.g. the diagram shows that the 100 percent rate of increased is reached at 2^2 coordinators for 4 workers and 2^6 coordinators for 64 workers.

In a distributed setting, the communication overhead is higher, and thus the overhead is increased.

4.3.3. Probability of Failure

Figure 10 illustrates the increase in the overhead that results from an increase in the probability of failure, i.e. the likelihood of the coordinator's constraints preventing a message from being processed. For low probabilities of failure, the impact on the overhead appears to be negligible. By contrast, the impact for high probabilities of failure is considerable.

The latter result should not come as a surprise. Doubling the probability of failure means that, on average, a message consideration fails twice as often. An 80 percent probability

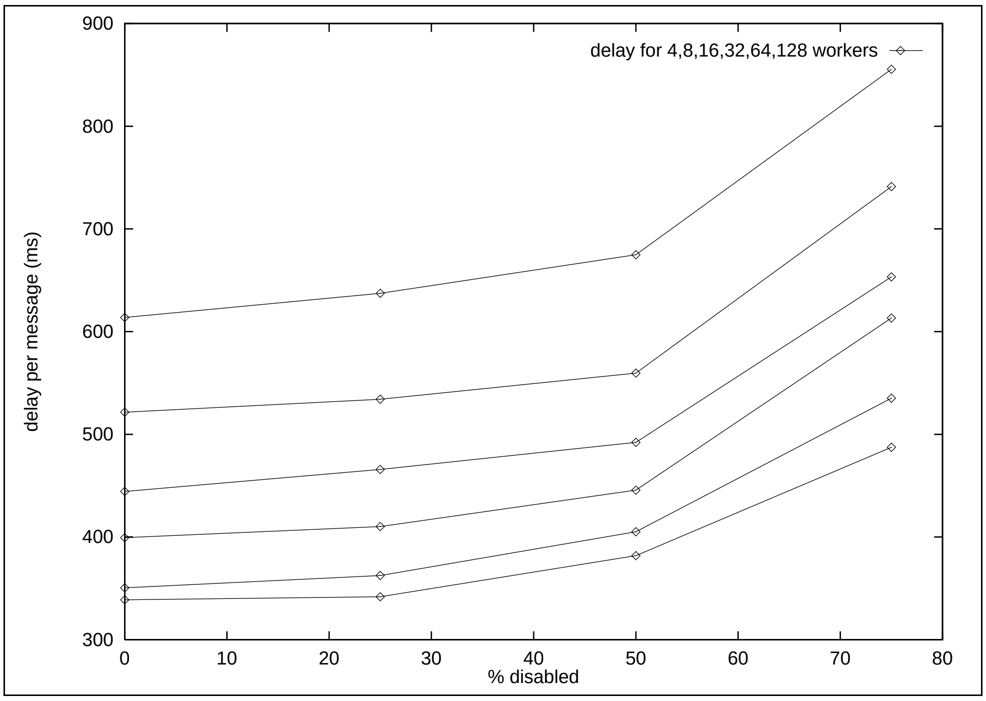

Figure 10. Dependency of the overhead on the probability of failure

of failure, for example, means that the consideration of a message fails approximately five times whereas a 40 percent probability only results in two-and-a-half failures. However, from this explanation we would expect a near linear curve, but instead the actual result curve is extremely flat for low probabilities and very steep for high probabilities. Tracing of the execution revealed the following: In our coordination algorithm a message consideration cannot be both aborted and failed, and aborts have a higher priority than failures. This means that messages can be aborted when they would fail otherwise. For low probabilities of failure this could take place almost all the time and message consideration would therefore almost *never* fail. With the probability of failure increasing, the chances of all failed message considerations being masked by aborted message considerations decreases, resulting in the steeper curve for high probabilities of failure.

In a distributed setting, the result will look similar, apart from the overhead being generally higher due to do the increased cost of communication.

4.4. Performance Evaluation Results

Summarising the results from above, we can make the following observations about the per-message overhead:

- The overhead resulting from the number of traps installed per component is negligible.
- The overhead resulting from the constraint evaluation is negligible.
- The overhead resulting from the use of reflected mailboxes is about two times that of using ordinary mailboxes.
- The overhead resulting from the implementation of coordination controllers in terms of reflective mailboxes is about twenty times that of using ordinary mailboxes.
- The overhead resulting from the workload increases linearly with the workload.
- The overhead resulting from the degree of coordination increases linearly with the number of coordinators.
- The overhead resulting from the probability of failure is negligible for low probabilities and substantial for high probabilities.

The results show that the base cost of coordination, i.e. the fixed overhead, is considerable. It could be reduced significantly by implementing all or most of the coordination support layer in C++ eliminating most of the cost of reflection and speeding up coordination-related computation.

Even with most of the coordination implemented in C++, some sizeable overhead will always remain and factors like the number of traps and the cost of constraint evaluation will become more dominating. These overheads can have a major impact on performance if coordination is performed at a fine-grain component level where the cost of a component processing a message is small. Coordination on this level typically does not have the requirement for separation, openness and adaptability which our model was designed for. Hence it is reasonable to perform coordination in a different manner, e.g. by making it part of the computation process. By contrast, the fixed cost of coordination (after optimisation of the implementation) is likely to be insignificant when coordination is performed on a coarse-grain component level, since there most of the overall cost of processing a message results from the associated computation.

The negligible impact of the number of traps and constraint evaluation allows us to further simplify the formula that calculates the per-message overhead.

$$t'_{message} = (n_{abort} + n_{fail} + n_{coord})(t'_{vote} + t_{roundtrip}) + n_{coord}(t'_{ack} + t_{roundtrip})$$

We can plot $t_{message}$ in a three-dimensional space for a given setting of n_{fail} (e.g. zero), with n_{work} and n_{coord} representing the x and y coordinates respectively. The results of this are shown in Figure 11. Our tests showed that n_{abort} is proportional to n_{work} and hence we would expect a plane, which is indeed what we get. This confirms the correctness of our formula for calculating the overhead.

In the above formula, the increase in the overhead due to an increase in the number of workers is constant. Specifically, it is independent of the number of coordinators. This will

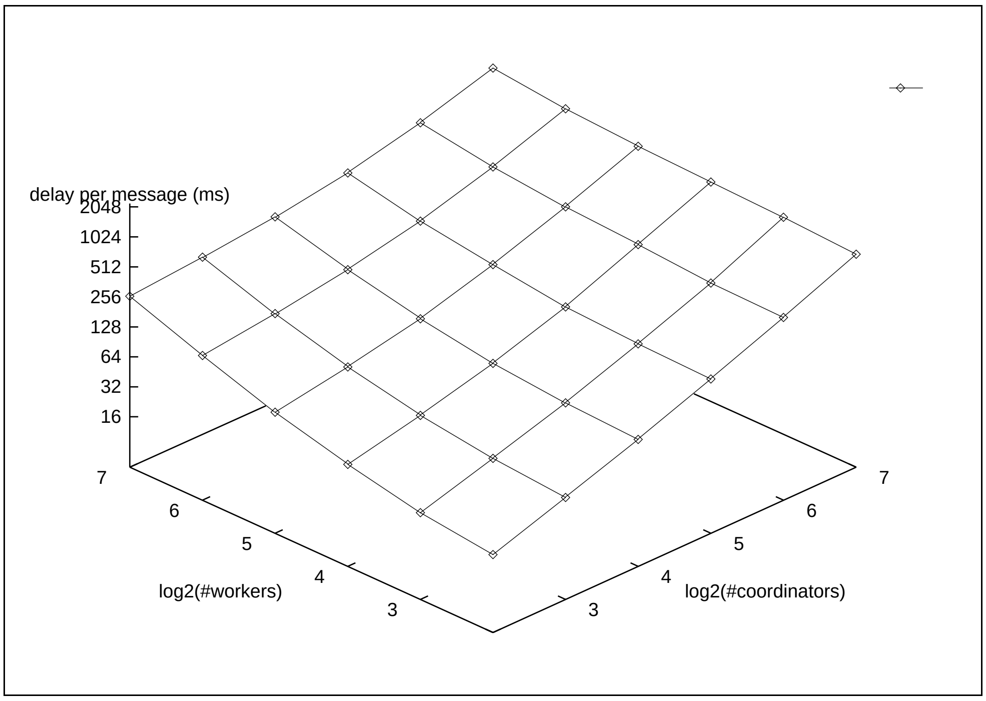

Figure 11. Dependency of the overhead on the workload and degree of coordination

not be the case in general since this independence is primarily due to the fact that in our test framework all workers are coordinated by the same coordinators and that the same constraints are enforced by all coordinators. In the worst case scenario, the *last* rather than the *first* coordinator will detect deadlocks (and cause an abort) and constraint violations (and cause a failure). This means that $t_{message}$, in the worst case scenario, is actually calculated as follows:

$$\begin{aligned} t_{message} &= (n_{abort}n_{coord} + n_{fail}n_{coord} + n_{coord})(t_{roundtrip} + t'_{vote}) + \\ &\quad n_{coord}(t_{roundtrip} + t'_{ack}) \\ &= n_{coord}(n_{abort} + n_{fail} + 1)(t_{roundtrip} + t'_{vote}) + \\ &\quad n_{coord}(t_{roundtrip} + t'_{ack}) \end{aligned}$$

Thus, the overhead is still linearly dependent upon n_{abort} and hence the number of workers, but the rate of increase per worker depends on n_{coord}.

The linear increase of the overhead with the number of workers means that a system is scalable, i.e. can cope with an increased workload by (linearly) increasing the number of worker nodes, provided that the following condition holds

$$t_{process} \gg n_{coord}(t_{roundtrip} + t'_{vote})$$

i.e. the time it takes a worker to process a message must be significantly greater than the differential of $t_{message}$ with respect to n_{work} (i.e. n_{abort}). We can calculate the *degree of scalability*

$$\mathit{scalability} = 1 - n_{coord}(t_{roundtrip} + t'_{vote})/t_{process}$$

i.e. a scalability of 1.0 means that in order to cope with a doubling of the workload the number of workers needs to be doubled, 0.5 means it needs to be quadrupled, etc.

The result has three important implications:

- Due to the linear dependency of the degree of scalability on the average roundtrip time, coordinators should be placed on the same node as the coordinated components or on node that is close in the network topology.[||] Practically this is actually almost always impossible, since a system that is scaled by adding nodes will not be placing worker components on the same nodes as coordinators.

- Efficient implementations of our model, that minimise the residual voting overheads, can greatly improve scalability, as there is a linear dependency of the degree of scalability on it.
- Minimising the degree of coordination, i.e. minimising the number of coordinators interested in any particular intercepted message, is essential, since there is a linear dependency of the degree of scalability on the number of coordinators — doubling the number of coordinators doubles the overhead.

The second item we already discussed above. We estimate that optimising our implementation on the Rosette level will reduce the overhead by a factor of 2-3. Re-implementing time-critical parts of our solution in C++ should reduce the overhead by a further factor of approximately 5, which is the typical performance gain achieved by moving code from Rosette to C++. This means that the base overhead of coordination will only slightly increase message delivery time. Furthermore, the degree of scalability in settings with a small number of coordinators and short message roundtrip times (which is precisely the types of system we are primarily interested in), will be close to 1.0 and hence application programmers will be able to utilise our coordination model in such a setting without having to worry about performance.

[||]This conclusion is really obvious; we include it for completeness.

Of most interest to the application programmer, is the last item. It implies that generally they should aim to implement any particular piece of coordination logic with as few coordinators as possible. However, there might be good reasons for selecting an algorithm that requires a higher number of coordinators per message. For instance, we could have coordinated the Dining Philosophers with a single "central" coordinator, in which case every message would only be subject to coordination by one coordinator instead of two (the chopstick's coordinator and one of its neighbours). Doing so, on the other hand, would actually make the solution unscalable since the workload on the single coordinator would increase with every philosopher added to the system. By contrast, the workload on coordinators in our implementation of the Dining Philosophers only depends on the workload of the neighbouring philosophers. A new coordinator is added with every new philosopher and can be placed on a separate node, thus permitting the scaling of the system. Note though that this scalability of the coordination logic, i.e. the ability of *coordinators* to cope with an increased workload, is only of concern if the amount of computation performed by the coordination logic is within the same order of magnitude or greater than the computation performed by the coordinated components. For the application programmer, finding the most efficient algorithm for implementing the coordination logic in such applications requires striking the right balance between the scalability of the computation layer (i.e. the coordinated components) and the scalability of the coordination layer (i.e. the coordinating components).

5. Conclusions

The model of coordination presented in this paper enables the coordination of components in open adaptive systems, independently from the underlying distributed system platforms. Some recent advances in this direction can be found in the notions of synchronisers [11, 10], regulated coordination [19] and programmable coordination media [9]. Our approach shares many of the initial motivations and there are also some similarities between the concepts. For instance, the notion of a programmable coordination media is based on intercepting messages in a very similar way to our traps, and underlying synchronisers is a coordination protocol that is in many ways similar to ours. However, in our opinion these models are still not sufficiently open and only have very limited support for system evolution and the abstraction and reuse of coordination patterns in a truly open setting.

We can make several important observations on the model presented in this paper. Firstly, the presence of coordinating components is transparent to the components in the computation layer that are being coordinated. Coordination can be imposed without changes to these components by observing and coercing their visible behaviour. All that is required is the ability to observe (and intercept) the messages emitted from components. Secondly, coordination will only take place where it is needed. Components are free to interact with each other without the involvement of the coordination layer if the coordination layer hasn't specified that such an involvement is required, by defining suitable traps. The safety and liveness requirements of a system can be met using whatever information is available about the messages sent/expected by a component, it's internal behaviour, protocols etc. This, somewhat pragmatic, approach enables the integration of components implemented in a multitude of languages and running on a multitude of systems.

Finally, we can observe that coordination in our model is performed by ordinary components residing in the computation layer. This is a result of the reflective nature of traps. The only difference is in the *role* played by the components. Their specification, design and implementation can utilise the same tools, paradigms and languages. Hence the means of abstraction and reuse apply to our coordination logic in the same way as they are applicable to the application logic. For instance it doesn't take much effort to abstract generic resource allocation coordination patterns from our Dining Philosopher example. In addition to the obvious software engineering advantages of implementing coordination logic in the same way as application logic, we gain the ability to perform meta coordination, ie. coordinator components themselves can be subject to coordination by other components. Thus, instead of statically categorising components into those dealing with configuration/coordination and those dealing with computation, we have a dynamic relationship between components that is a result of the role they play with respect to each other at a particular point in time. This dynamic categorisation provides the means for implementing systems where coordination is an integral part of the functionality, and hence complex interactions take place between the coordination and computation layers.

Traps can easily be integrated into existing systems by modifying the communication layer. Thus all existing application code remains unaffected and the model functions in a heterogeneous setting, enabling the coordination of existing components across system boundaries. We have successfully implemented trap-based coordination in a heterogeneous setting consisting of a distributed actor-based system and a CORBA half-bridge [30]. The model has been successfully used commercially by TECC Ltd in the design and implementation of several middleware applications [29]. Our current research concentrates on establishing a formal semantics for traps and their application in the definition of reusable coordination patterns.

6. Acknowledgements

We gratefully acknowledge the advice and help provided by the Distributed Software Engineering Research Section at the Imperial College Dept. of Computing and the members of the Middleware Systems Group at TECC Ltd, as well as the financial support from the EPSRC under grant ref: GR/K73282.

REFERENCES

1. F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer-Verlag, April 1996.
2. J.-P. Banatre and D. Le Metayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.
3. M. Banville. Sonia: an adaption of Linda for coordination of activities in organizations. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer-Verlag, April 1996.
4. J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer-Verlag, April 1996.
5. E. Black, J. Fournier, and P. Kastner. *Enterprise Application Integration: Advanced Technologies and A Sense of Process*. Aberdeen Group, 1998. `http://www.aberdeen.com`.
6. G. Blair and M. Papathomas. The case for reflective middleware. Proc. of the 3rd Cabernet Plenary Workshop `http://www.newcastle. research.ec.org/cabernet/workshops/3rd-plenary.html` , April 1997.
7. G. Blair and J.B. Stefani. *Open Distributed Processing and Multimedia*. Addison Wesley Longman, 1997.
8. C.J. Callsen and G. Agha. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.
9. E. Denti, A. Natali, and A. Omicini. Programmable coordination media. In D. Garlan and D. LeMetayer, editors, *Coordination Languages and Models, 2nd Int. Conference*. Springer-Verlag, September 1997.
10. S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

11. S. Frolund and G. Agha. A language framework for multi-object coordination. In *ECOOP'93 Proceedings*, volume 707 of *LNCS*. Springer-Verlag, 1993.
12. D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proc. of the 16th Int. Conf. on Software Engineering*, May 1994.
13. D. Garlan and M Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Co., 1993.
14. David Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, 1985.
15. A.A. Holzbacher. A software environment for concurrent coordinated programming. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, 1st Int. Conference*, volume 1061 of *LNCS*. Springer-Verlag, April 1996.
16. G. Kiczales, D. Gureasko, and J. Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.
17. J. Magee, S. Eisenbach, and J. Kramer. System structuring: A convergence of theory and practice? In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, Proc. of the Dagstuhl Workshop*, volume 938 of *LNCS*. Springer-Verlag, 1995.
18. S. Matsuoka, T. Wanatabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In O. Nierstrasz, editor, *ECOOP'91 Proceedings*, LNCS. Springer-Verlag, 1991.
19. N. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In D. Garlan and D. LeMetayer, editors, *Coordination Languages and Models, 2nd Int. Conference*. Springer-Verlag, September 1997.
20. T. Mowbray and R. Zahavi. *The Essential CORBA: Using Systems Integration, Using Distributed Objects*. John Wiley & Sons, 1995.
21. D.E. Perry and A.L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
22. A. Pope. *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison-Wesley, 1997.
23. N. Pryce. A model of interaction in concurrent and distributed systems. In *Second International Workshop on Development and Evolution of Software Architectures for Product Families*, February 1998.
24. J.M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages*, 16(1):151–174, January 1994.
25. M. Radestock and S. Eisenbach. Agent-based configuration management. In *Proc. of the 7th IFIP/IEEE Int. Workshop on Distributed Systems: Operation and Management*, 1996.
26. M. Radestock and S. Eisenbach. Formalizing system structure. In *Proc. of the 8th Int. Workshop on Software Specification and Design*, pages 95–104. IEEE Computer Society Press, 1996.
27. R. Rock-Evans. *Ovum Evaluates: Middleware*. Ovum, 1997. `http://www.ovum.com`.
28. A. Schill, editor. *DCE — The OSF Distributed Computing Environment*. Springer-Verlag, October 1993.
29. F. Taylor and M. Radestock. TECCware product definition. Technical report, Trans Enterprise Computer Communications Ltd, 1998. `http://www.tecc.co.uk`.
30. S. Eisenbach, E. Lupu, K. Meidl and H. Rizkallah. Can Corba Save a Fringe Language from Becoming Obsolete?. DAIS99 Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, Helsinki, June 1999.