Coordination in Evolving Systems

Matthias Radestock and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ
E-mail: {mr3,se}@doc.ic.ac.uk

Abstract. To facilitate the writing of large maintainable distributed systems we need to separate out various concerns. We view these concerns as being communication, computation, configuration and coordination. We look at the coordination requirements of long running systems, paying particular attention to enabling the dynamic addition and removal of services. We show that the key to a smooth integration of configuration and coordination into systems is a new style of communication. We show how these ideas can be incorporated into the actor model.

1 Introduction

When designing and implementing large-scale heterogeneous distributed systems the software engineer faces challenges that would not be encountered in sequential programs. The problem of *coordination* [MC94], forms a central part of the challenge – the activities of the system components need to be coordinated such that the overall system behaviour conforms to the specification. Coordination is an issue that arises *after* a range of other problems in a distributed system have been tackled. These include the distribution of components, the communication protocols, the data exchange between different platforms, fault tolerance, and migration. Typically these issues are addressed by *distributed systems platforms*, such as CORBA[MZ95]. Application design and implementation should not need to be concerned with them, apart from *using* the provided mechanisms. By contrast, coordination is often entirely embedded in the application design and implementation, and is not treated as a separate concern. This ignores the fact that coordination is a very distinct aspect of distributed systems, and that it contains application independent elements as well as elements that may be common to several applications.

To isolate the coordination elements, we can split the specification and implementation into four parts – communication, computation, configuration and coordination. Communication deals with the exchange of data, with a foundation of communication paradigms such as request-reply, synchronous and asynchronous. Computation is concerned with the data processing algorithms required by an application, with a foundation in traditional paradigms such as functional programming and object-oriented programming. Configuration determines which system components should exist, and how they are inter-connected, and is based

on principles of *software architecture*[GS93, PW92, GP94]. Finally coordination is concerned with the interaction of the various system components, and is founded on recent paradigms such as *process calculi*[Mil89, MPW92, Mil91] and the notion of *interaction machines*[Weg96].

Having to perform three paradigm shifts during the design and implementation of a system is costly, since ultimately all elements have to work together to meet the overall system specification. In addition to the difference in paradigms, each element may use its own specification and implementation language. The software engineer thus potentially has to deal with four paradigms, four specification languages and four implementation languages. This makes system design complicated, analysis almost impossible, and maintenance expensive. A much preferred scenario would enable us to work within one single framework without loss of generality, ie. without losing the ability to integrate various specification and implementation languages. In this paper we argue that an enriched actor model can be the underlying paradigm for this.

We analyse the requirements of configuration and coordination and their impact on the requirements for an implementation. We introduce the notion of *implicit anonymous communication* as the key idea with which these requirements can be achieved, whilst integrating transparently into existing models. Thus from the point of view of computation, coordination does not exist. We then look at the actor model as a basis for integration and show that computation and communication concerns are already integrated into it. The model, as it stands, is not suitable for configuration and coordination, therefore we propose extensions that enable their integration. The extensions are rooted in the existing model by changing the semantics of communication. They are thus totally transparent which enables us to *add* configuration and coordination to existing designs without the need to change these.

2 Requirements of Coordination

We can view a distributed system as a collection of distributed agents[1] that interact with each other. The concerns of a distributed system can be separated into four parts (cf. Fig. 1):

- The *communication part* defines *how* agents communicate with each other.
- The *computation part* defines the implementation of the *behaviour* of individual agents. It thus determines *what* is being communicated.
- The *configuration part* defines the *interaction structure*, or *configuration*. It states which agents exist in the system and which agents 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.

[1] We use the term *agents* in a somewhat lax way as denoting an entity that 'does something'.

The communication part is the only part that is totally application independent, and thus features in the design and implementation as something that is being *used*, rather than defined or altered. The computation, configuration and coordination parts all include application dependent elements. However, each of them also has its own set of general, application independent requirements. In addition to this, inter-part dependencies yield a layered system structure. The coordination layer depends on the configuration layer because it requires information about the interaction structure in order to determine possible communications. The configuration layer depends on the computation layer since it needs to know which kinds of agents have been defined in order to be able to create new agents and to establish with which other agents an agent can communicate. The computation layer depends on the communication layer for the exchange of information. The layered structure also means that lower layers need

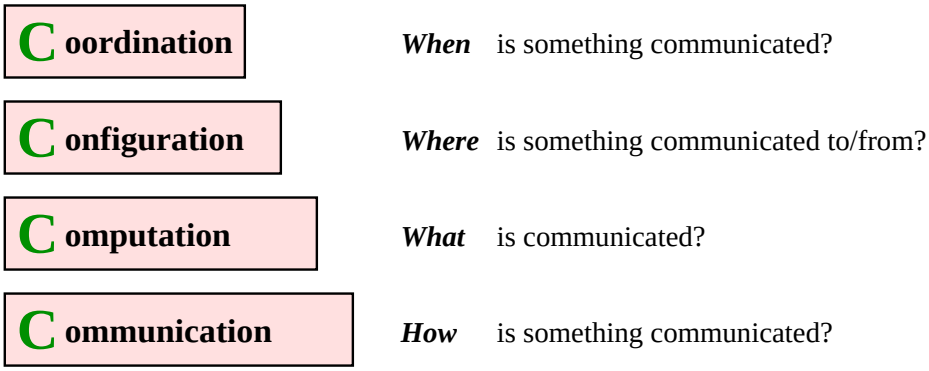

Fig. 1. The four layers of a specification of a distributed system

not, and should not know about the higher layers – as far as the lower layers are concerned the upper layers need not exist. This clear separation of concerns is extremely beneficial, enabling a high degree of reuse and easier maintenance. The aim of our research is to combine the layers in one framework without compromising their separation. This uniformity considerably reduces design and implementation time since the same methods, principles and tools can be applied to all layers. It also makes it easier to describe the inter-layer dependencies.

Before devising an integrated framework we need to investigate the requirements posed by each of the layers. The requirements of the communication and computation layer are quite well understood. By contrast, the requirements of configuration and coordination have so far received little attention. Since the coordination layer depends on the configuration layer, we first analyse the requirements of the latter and then look at the specific requirements of coordination.

2.1 From Static to Dynamic Configuration

In simple distributed systems a fixed set of interactions takes place between a fixed set of agents. The interaction structure thus only needs to be established once, at system start-up. No interaction between the computation part and configuration part is required after that, in fact, the configuration part need not even exist anymore. We shall call such a model of configuration *static configuration*. Needless to say, these systems cannot accomodate any form of dynamic change, ie. they cannot respond to a changing environment or changing requirements. In fact they cannot even cope with change if it is part of the requirements. So why bother with such a model of coordination at all? The reason is that the vast majority of distributed applications include elements of such a static nature. Typically they appear at coarse-grain levels of decomposition in the design stage. So, for instance, a video conferencing client may consist of a video camera, a microphone and a screen. These components will always exist and connections between them are fixed.

At a more fine-grain level of decomposition the interaction structure within a system changes dynamically: new agents are created, existing agents are destroyed, connections between agents 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 agents in the system themselves, and so the configuration layer needs to exist during the entire life time of the system. We hence require an interaction mechanism between the computation part and the configuration part. The configuration part must have a run-time representation in order to enable dynamic access to its functionality. We shall call this model of configuration *dynamic configuration*. It subsumes the static configuration model.

2.2 From Configuration to Coordination

Coordination specifies patterns of interaction. Such a pattern may, for instance, be that the agent A can only send message X to agent B after agent C has sent message Y to agent D. Coordination requires configuration since before specifying the patterns of interaction, the parties of the interaction need to be specified – which is precisely the problem addressed by configuration.

Traditionally interaction patterns have not been specified explicitly, but were an implicit element in the design and implementation of the computation part. This makes it difficult to check and enforce adherence to the specification, limits the reuse of the thus constructed agents and complicates maintenance. To overcome these difficulties, the coordination layer should have an explicit representation, making it possible to *specify* the interaction patterns. We can also 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 the satisfaction of the application specification, ie. the changes to the structure are ultimately

triggered by computational agents. So a mechanism is required that enables the interaction with both the configuration and computation part.

To enforce the adherence to the specification in a running system the coordination part needs to be able to observe and interfere with interactions. Thus, although the interaction patterns in a static configuration model are fixed throughout the life-time of the system, the coordination part must exist at run-time.

2.3 Interactive 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 computational agents. However, this is insufficient in many large distributed systems, especially multimedia systems. Such systems are often long-lived, needing to be kept running for days and in some cases even years. These systems require interactive management – human agents need to be able to reconfigure the system while it is running. Furthermore they 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 agents representing the screen need to be added to the system's computation layer. Then the configuration layer needs to be modified to forward all data of the conferencing communication to that agent. Finally we need to alter the coordination layer to ensure that the new agent interacts with the rest of the system in the desired manner. We shall refer to such systems as *interactive systems*. They are capable of accomodating 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.

Interactive systems require an explicit and tangible run-time representation of the computation, configuration and coordination parts, since we need the ability to modify them interactively. This is the only difference in requirements from dynamic systems.

2.4 Summary

The increasing complexity of requirements when increasing the dynamism, ie. moving from static to dynamic and finally interactive systems, can be observed for all four layers. A system can include a mixture of static, dynamic and interactive layers; however, lower layers require at least the same degree of dynamism from higher layers, as we illustrated above. Also, application requirements usually result in a varying degree of dynamism for different parts of the specification and implementation.

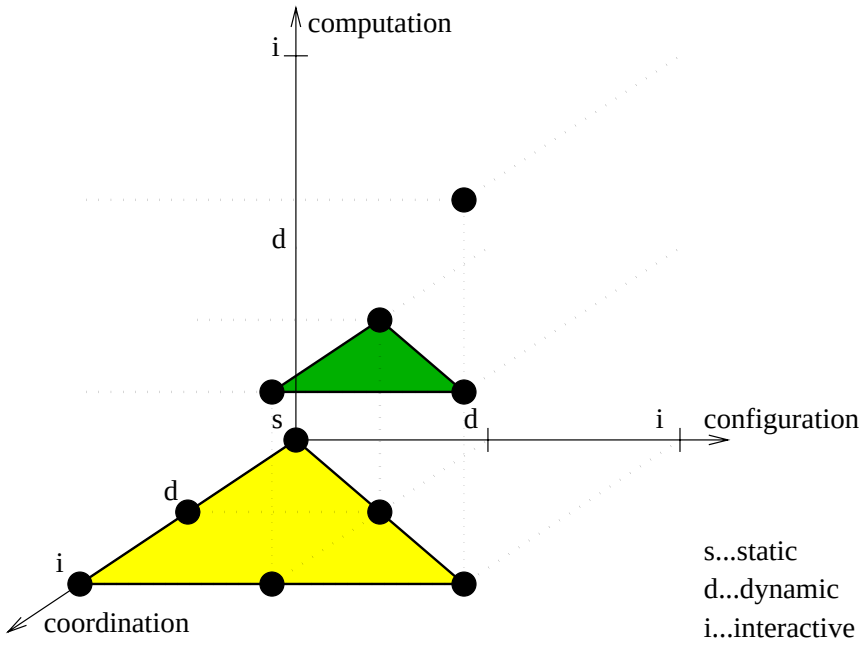

Fig. 2. Classification of Distributed Systems According to Layer Dynamism

Fig. 2 illustrates the classification of distributed systems according to the degree of dynamism in the various layers. Since the communication layer is application independent it does not contribute to the classification scheme. The lower plane contains systems with at least one static layer. The middle plane contains systems with no static, but at least one dynamic layer and the highest plane (which is in fact just a point) contains systems with only interactive layers. Additional requirements arise in a system whenever the dynamism increases in any layer. Thus systems with interactive computation, configuration and coordination are the most demanding. They require

- *Dynamic layers.* The ability to dynamically create new kinds of agents and modify their behaviour, and the ability to dynamically alter the interaction structure and interaction patterns.
- *Communication interception.* The ability to observe and interfere with communication activities in a system.
- *Layer interaction.* The ability of interaction between the coordination layer, configuration layer and computation layer.
- *Tangibility.* Explicit, tangible run-time representation of all layers.

In addition to these functional requirements there are also requirements derived from general software engineering principles:

– *Reuse.* Methods of abstraction and decomposition need to exist that enable the reuse of parts of the specification and implementation in different contexts.
– *Integration.* The ability to apply the same design and implementation principles and methods to any of the layers.
– *Uniformity.* Providing a uniform view of static, dynamic and interactive aspects.

3 The Key to Coordination

When investigating the requirements of coordination we discover a range of apparent conflicts between the above requirements. It is these conflicts that make coordination such a difficult problem to tackle, and we can find the key to coordination in the resolving of those conflicts.

Configuration is concerned with determining which agents can communicate with each other. Principally there are two ways of achieving this – either agents are told by the configuration part with whom they can communicate and use *non-anonymous communication*, or agents can use *anonymous communication* which is made concrete by the configuration part. Both approaches have drawbacks.

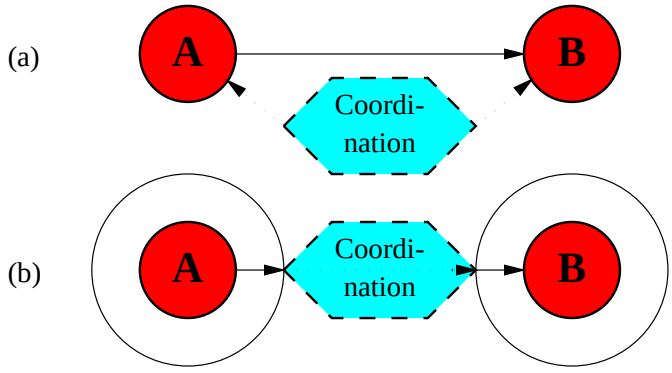

Fig. 3. Non-Anonymous and Anonymous Communication

In non-anonymous communication (Fig. 3a) messages contain a reference to a target agent. The target agents can be determined by the configuration layer, usually at instantiation time. However, this approach conflicts with the exchange of agent references in interactions, since this is a way that an agent can acquire the reference of another agent (and thus potentially communicate with it), without involvement of the configuration part. Furthermore the configuration becomes highly dependent on the implementation since it needs to know which

agent references a particular agent needs to be supplied with. Specifying this, for instance in terms of roles and using some type system, is complex.

In anonymous communication (Fig. 3b) no target is specified as part of the messages. The target(s) is(are) determined by the configuration layer. This approach suffers from a lack of control on the part of the agent, since an agent cannot ensure, or even indicate that two separate interactions should take place with the same destination agent. Anonymous communication is incompatible with function call and method invocation style programming, in the sense that such types of interactions carry a substantial overhead if they are to be modelled using anonymous communication. Anonymous communication also *requires* a configuration layer, otherwise no communication is possible at all. This makes the computation layer highly dependent on the configuration layer.

3.1 Implicit Anonymous Communication

We can address the problems of anonymous and non-anonymous communication by introducing a new style of communication – *implicit anonymous communication*. The idea is to *apparently* allow agents to send messages to other agents. However, these messages will *actually* be intercepted by the coordination layer and an appropriate action will be taken, possibly sending the message to the specified agent, or even to some other agents, unknown to the sender (cf. Fig. 4). The anonymity of the communication is thus implicit – to the agent it looks like a non-anonymous communication. Explicit anonymous communication is achieved by the agent by addressing the message to itself.

Fig. 4. Implicit Anonymous Communication

As part of the implicit anonymous communication model we intercept messages and forward them to agents, thus satisfying the communication interception requirement. The intercepted messages can be sent to agents dealing with configuration or coordination, whose main task is to alter the interaction structure and interaction patterns. Thus the dynamic layer requirement is satisfied. The integration requirement is satisfied by the very fact that configuration and coordination is performed by agents. Whatever principles and methods exist for the design and implementation of the computational agents can be applied to the configuration and coordination agents. Hence we can also satisfy the reuse requirement, provided that the framework we use for the design and implementation of our agents has sufficient support for it.

Integration also means that configuration and coordination agents are subject to configuration and coordination. We can thus create meta layers of configuration and coordination. The layer interaction requirement is in turn satisfied by integration – agents in lower layers can send messages to agents in higher layers and thus achieve explicit interaction. More commonly the interaction is implicit – as the result of an intercepted communication agents in higher layers send messages to agents in lower layers. Since configuration and coordination is performed by agents we have an explicit run-time representation and thus satisfy the tangibility requirement, provided that the implementation in general has explicit run-time representations of agents. The behaviour of the agents determines how configuration and coordination takes place and this behaviour can be altered at run-time, provided the model and implementation allow the alteration of agent behaviour in general.

The most striking feature of implicit anonymous communication is that its integration into a model is transparent. As far as the rest of such a model is concerned, nothing has changed – configuration and coordination apparently does not take place. This enables the integration of configuration and coordination into existing designs by way of *adding* it without having to change any existing parts of the specification.

3.2 Structural Reflection and Interpretation

The only requirement that is not addressed by implicit anonymous communication is the uniform view of static, dynamic and interactive aspects. By that we mean that the methods and principles used for designing and implementing configuration and coordination should be independent from the dynamism of the layers. As far as the model is concerned, dynamic systems subsume static systems. Interactive systems require the availability of the layers at run-time, but otherwise seem no different from dynamic systems – the issue is *how* the layers can be accessed at run-time.

Coordination agents require access to other coordination agents since, as part of their required functionality, they must be able to inspect and subsequently modify the coordination part. The same is true of configuration agents. In the case of dynamic layers this can be achieved by providing suitable accessors, since it is known at the design stage what information about the layer needs to be gathered at run-time. However, for interactive layers a more sophisticated approach is required, as we need to have the ability to inspect and modify *any* part of the the layer, without knowing this at the design stage. Since configuration and coordination are performed by agents, this can be achieved by providing a general agent inspection mechanism.

There are two approaches to agent inspection: we could extend our model with all the necessary functionality for agent inspection, or we could employ structural reflection. The first solution has obvious drawbacks since it can make the model substantially more complex. Structural reflection[JA92, MWY91], on the other hand, has a minimal impact on the model. The idea is to make the meta-level architecture identical to the architecture described by the model. In

other words, we describe the structure and functionality of agents in terms of other agents. The agent representation of the structure and functionality can then be inspected using the functionality provided by that agent. Note that this approach also simplifies the design and implementation of dynamic configuration and coordination, since no special accessors need to be defined anymore.

Structural reflection is only one part of the solution to accessing the configuration and coordination parts at run-time – it achieves uniformity as far as the model is concerned. However, this does not imply uniformity in the implementation. The latter is characterised by uniformity in the means by which the model is exploited for both dynamic and interactive configuration and coordination, ie. the inspection of an agent should 'look' the same no matter whether it is done dynamically or interactively. This can be achieved by making the implementation interpreter based. Thus, whatever constructs we use in the implementation for inspecting and modifying agents dynamically we can also use interactively. It turns out that interpretation is also required for interactive computation layers – in order to create new kinds of agents or substantially alter the behaviour of existing agents. An interpreter based implementation enables an incremental design where new kinds of agents are added to the system in precisely the same way as the agents that exist initially.

4 The Actor Model

The actor model[Agh86] is a simple yet powerful means of defining agent based systems. An actor is an entity that is represented by a *reference* and a current *behaviour*. Each actor represents an independent active entity, thus resulting in inter-actor concurrency. The basic form of interaction between actors is buffered asynchronous peer-to-peer communication. Thus, associated with every actor is a *mail queue* that serves to buffer messages that are sent to the actor until they can be accepted for processing. When an actor starts processing a message it is *locked* until a *replacement behaviour* is established that will take over the processing of the next message. Since this can take place before the processing of the message has been completed, a form of pipelined intra-actor concurrency is achieved. The behaviour of an actor determines the actions to be taken in response to a message. It typically includes a set of *acquaintances*, which are references to other actors. The content of a message may also contain references, and together with the acquaintances, they comprise the set of known actors. Fig. 5 illustrates the structure of an actor. There are two types of actions an actor can take in response to a message: the sending messages to known actors and the creation of new actors. Primitive actors are used in the model to avoid a conceptually infinite regress of message passing. An implementation will give direct treatment to passing messages to primitive actors.

4.1 Computation and Communication

All true computation ultimately takes place in primitive actors which have a built-in behaviour. Non-primitive actors use these primitive actors by sending

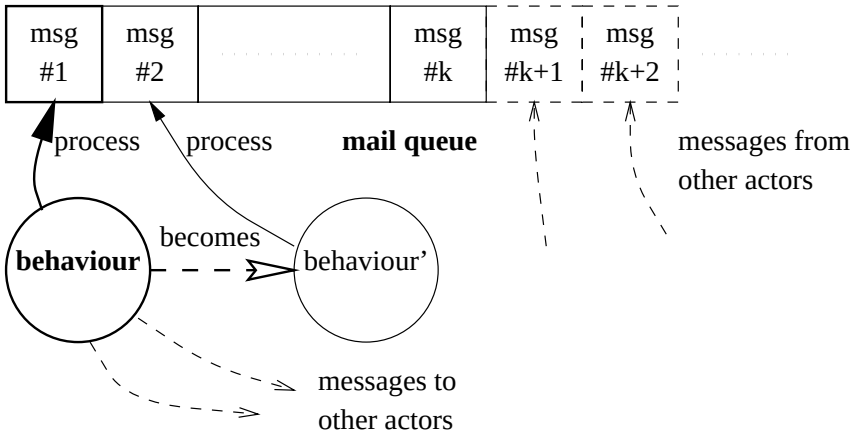

Fig. 5. The Actor Model

messages to them. In doing so they establish complex behaviours and thus perform complex computations. The buffered asynchronous peer-to-peer communication is an intrinsic feature of the model and underlies computation. Other forms of communication can be built on top of it. Causality, for instance, can be achieved using request-reply style interactions, which in turn can be implemented by embedding 'self' references in messages.

The majority of core problems of distributed systems can be addressed by implementations of the model, and thus need not be of any concern to application designers and implementors. For instance, there is nothing in the actor model that prevents distribution of actors. All implementations need to do is to give some distributions semantics to actor references. Data exchange between different platforms and actor-migration can be achieved in a similar way, by giving more concrete semantics to message passing and actor references. Note that this is precisely the way primitive actors are supported – the application developer need not have any knowledge about whether an actor is primitive.

4.2 Configuration and Coordination

The basic actor model is unsuitable for purposes of configuration and coordination because it relies on the asynchronous peer-to-peer communication. Actors can only send messages to actors they know. This knowledge is embedded in the actors, whereas it should really reside in the configuration layer. Configuration can only be achieved to some degree by having the configuration part 'tell' the agents with whom they can communicate. The same approach would have to be taken for coordination. By altering the semantics of communication in the model we can introduce implicit anonymous communication – a message sent by

an actor to another actor thus not automatically ends up in the target's mail box but may instead be 'diverted' to another actor which performs configuration and coordination tasks.

Structural reflection is consistent with the actor model. An actor consists of just three entities: a mail queue, a set of acquaintances and a behaviour. By introducing method and procedure actors we can describe the behaviour in terms of actors. The behaviour can thus be unified with the set of acquaintances – an actors behaviour is determined by the procedures and methods it 'knows'. The mail queue can be viewed as a special acquaintance. The set of acquaintances can be viewed as a list of key-value pairs, so-called *slots*. Acquaintances can be identified by their key, which in turn refers to an actor. Hence an actor can be described in terms of a list containing actor references, which in turn, can be viewed as an actor. Messages can also be viewed as actors, containing a slot that refers to the target of the message and a slot that holds the content, for instance, in the form of a tuple. Thus everything in the actor model can be described in terms of actors and we can have complete structural reflection.

5 Related Research

Coordination is a relatively new research topic. Nevertheless, several coordination languages and systems have been developed. Formalisms, such as Gamma[BM90] and languages such as Linda[Gel85, Ban96] have emerged. Most of these systems are not intended to be general-purpose design and implementation frameworks. They are proof-of-concept and theoretical systems. The clear separation of layers, their transparent integration and interactive nature of the layers have not been addressed. These are important issues when coordination is viewed as a software engineering issue. For the issue of configuration this perspective has been taken by research in software architecture[GS93, PW92, GP94, RE96]. The focus has only been on layer separation though, without paying much attention to transparent integration and without recognising coordination as a separate layer. Additionally the interactive aspects have been neglected and even dynamic layers are sometimes not supported. More recently, some attempts have been made to integrate coordination into such systems – ToolBus[BK96] is an extension of the PolyLith Software bus[Pur94], ConCoord[Hol96] is an extension of *Darwin*[MEK95]. ActorSpace[CA94] and Synchronizers[FA93, Fro96] are extensions of an actor language. Manifold[Arb96] is based on a model where processes communicate anonymously via streams.

The ToolBus architecture views a system as a collection of tools that communicate with each other via a bus. While this achieves a clear separation of the configuration and coordination layers from the computation layer it does not achieve their transparent integration. The computation layer has to be modified to allow configuration and coordination, which in that case are *required* in order for anything to happen in the system. Interactive layers are not supported. Only limited means of abstraction and reuse are available.

In ConCoord a distributed system is viewed as a collection of *components*

with *interfaces*. Interfaces represent services that are either provided or required. Consequently ConCoord distinguishes between provisions and requirements and it is the task of the configuration layer to establish *bindings* between the two. Communication is thus explicitly anonymous, with all the associated drawbacks. Components are either primitive or are compositions of other components, thus resulting in a hierarchical decomposition structure. Primitive components represent the computation layer. Coordination is triggered by *state notification* and results in reconfiguration. Significant changes to the coordination layer thus require alteration of the computation layer, which is not possible dynamically or interactively. Interactive layers are not supported in general. The expressiveness of the configuration language is limited and thus forces separate languages, design and implementation principles for the computation layer.

MANIFOLD is based on concepts very similar to those underlying ConCoord. The language is more powerful and in principle allows a unified design and implementation approach for the layers. However, it suffers from the drawbacks of using explicit anonymous stream-based communication as the only communication model. Events are used for layer interaction and also for interaction within the configuration and coordination layer. Configuration and coordination of the event-based communication is complicated and not explicitly supported. Interactive layers are not supported.

ActorSpaces provide an anonymous communication mechanism for actors. An actor can send a message containing a pattern which is matched against known ActorSpaces which in turn match it against the list of visible attributes of actors in their ActorSpace. The sender of the message can specify whether the message should be broadcast to all matching actors or whether it should be sent to one chosen actor. In essence, ActorSpaces determine the receiver(s) for messages and represent the configuration layer. Synchronizers constrain the invocation patterns on groups of actors by imposing temporal and causal orderings on the messages received by actors within the group. They thus represent the coordination layer. Synchronizers do not require alterations to the computation layer but ActorSpaces do. Both are not explicitly modelled as actors. Their behaviour is specified using concepts outside the actor model and thus requires new design and implementation methodologies. It also makes interactive layers impossible since no tangible explicit run-time representations of the configuration and coordination layer exist.

6 Summary

In this paper we have illustrated that there is a need for dividing the specification and implementation of distributed systems into four parts – communication, computation, configuration and coordination. We then showed that these four parts require integration into a single framework. An investigation into the dependencies between the parts reveals a layered structure where lower layers are unaware of higher layers. The coordination layer is the highest layer and as far as the other layers is concerned coordination thus does not exist. We discov-

ered that the requirements for an integrated framework largely depend on the dynamism present in each of the layers and that there is a an interdependency between the requirements. Systems with interactive layers were shown to be the most demanding, but are the only ones that can satisfy the requirements of complex, long-running distributed applications.

We have shown that the underlying communication model plays a crucial role in meeting the requirements of configuration, coordination and their integration into an overall framework. Both non-anonymous and anonymous communication prove to be unsuitable for a general solution. We introduced implicit anonymous communication as a new model of communication. It satisfies most of the requirements, with the remaining ones being met by structural reflection and an interpreter-based approach. Most importantly it turns out to be a means by which configuration and coordination can be integrated transparently into the overall framework. Thus, for instance, the layer's conceptual unawareness of the coordination layer is preserved in the model.

We have illustrated how implicit anonymous communication can be added to the actor model by changing the semantics of communication in the model. Structural reflection can also be added easily. This again allows the transparent integration of configuration and coordination. We believe that implicit anonymous communication can be *the* means by which configuration and coordination can be integrated transparently into existing models of distributed systems design and implementation. It is the foundation upon which various higher level configuration and coordination concepts can be based.

7 Acknowledgements

We gratefully acknowledge the advice and help provided by TECC (`http://www.tecc.co.uk`), the Distributed Software Engineering Research Section (`http://www-dse.doc.ic.ac.uk`) at the Imperial College Dept. of Computing, and the financial support from the EPSRC under grant ref: GR/K73282.

References

[Agh86] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Arb96] F. Arbab. The iwim model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[Ban96] M. Banville. Sonia: an adaption of linda for coordination of activities in organizations. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[BK96] J.A. Bergstra and P. Klint. The toolbus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[BM90] J.-P. Banatre and D. Le Metayer. The gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[CA94] C.J. Callsen and G. Agha. Open heterogeneous computing in actorspace. *Journal of Parallel and Distributed Computing*, pages 289–300, 1994.

[FA93] S. Frolund and G. Agha. A language framework for multi-object coordination. In *ECOOP'93 Proceedings*, volume 707 of *LNCS*, pages 346–360. Springer Verlag, 1993.

[Fro96] S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, 1985.

[GP94] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proc. of the 16th Int. Conf. on Software Engineering*, May 1994.

[GS93] 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.

[Hol96] A.A. Holzbacher. A software environment for concurrent coordinated programming. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[JA92] S. Jagannathan and G.A. Agha. A reflective model of inheritance. In *ECOOP'92 Proceedings*. Springer-Verlag, 1992.

[MC94] T. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26:87–119, March 1994.

[MEK95] 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.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, October 1991.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100:1–77, 1992. Also as Tech. Rep. ECS-LFCS 89-85/86, University of Edinburgh.

[MWY91] 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.

[MZ95] T. Mowbray and R. Zahavi. *The Essential CORBA: Using Systems Integration, Using Distributed Objects*. John Wiley & Sons, 1995.

[Pur94] J.M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages*, 16(1):151–174, January 1994.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[RE96] 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.

[Weg96] P. Wegner. Coordination as constrained interaction. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

This article was processed using the LaTeX macro package with LLNCS style