COMPOSING DISTRIBUTED RESOURCE MANAGEMENT ACTIVITIES

BY

NALINI VENKATASUBRAMANIAN

B.E., Bangalore University, 1989
M.S, University of Illinois, Urbana-Champaign, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

Abstract

Advances in networking, communication, storage, computing, and multimedia technologies coupled with many emerging application areas is fueling the merger of computing and communication systems. This will result in a global information infrastructure of the size and magnitude erstwhile unimaginable. Such an infrastructure will have numerous services and hundreds of thousands of subscribers. A key issue in developing a global information infrastructure is that of effective management and utilization of resources. Increasingly, applications require delivery of multifaceted digital information services with stringent requirements on the delivery of information. For instance, multimedia applications have QoS (Quality of Service) parameters that define the extent to which performance specifications such as responsiveness, reliability, availability, security and cost-effectiveness may be violated. Varying requirements posed by applications, customers, and service providers makes the task of resource management in the evolving global information infrastructure a challenging research problem - one with significant commercial impact as well.

In this thesis, we present a new paradigm for developing safe, customizable middleware for the global information infrastructure. The composition of multiple resource management services is necessary to guarantee safe, cost-effective QoS in such an infrastructure, which by its very nature is open and distributed. We specify core resource management services – remote creation, distributed snapshot and directory services that can be used as a basis for more complex activities. The thesis develops mathematical frameworks and formal mechanisms for reasoning about the interaction and composition of resource management activities in open distributed systems, their dynamic installation and modification. In particular, we develop a two-level meta-architectural model of distributed computation based on Actors. This enables us to consider separately issues such as: functional behavior of an application; and resource management issues such as storage management, load balancing, QoS specification and enforcement. The utility of this approach is illustrated by developing QoS based resource management techniques for distributed multimedia systems and reasoning about them.

To ....

Acknowledgements

It has been a privilege to work with Prof. Gul Agha, my thesis advisor. I shall always be grateful for his guidance, advice, numerous discussions, patient iterations of my early research and his confidence in my abilities. I also thank Gul for fostering an inspiring and flexible research environment by encouraging everyone in our research group to discover their interests and goals. I have utmost respect for his wisdom, his kind and gentle disposition and continue to be amazed by his ability to provide an overarching vision to various research areas.

I am extremely fortunate to have received the guidance of Dr. Carolyn Talcott at Stanford University, where a large part of this dissertation research was conducted. I would like to express my deep gratitude to Carolyn for being so generous with her time and help, and for the many stimulating discussions we have had over the years. I remain in awe of her attention to detail, thorough scrutiny of research, keen intellect, dedication and commitment. I have always looked forward to the Saturday research meetings with Carolyn, a welcome respite from the surrounding commercial Silicon Valley bustle and a way for me to not lose sight of the larger technical vision. I would also like to thank my other doctoral committee members, Prof. Klara Nahrstedt and Prof. Geneva Belford for invaluable advice and comments. Over the past two years, I have had the opportunity to collaborate with Klara on various aspects of QoS management. I would like to thank her for the many enjoyable discussions and for continually providing new perspectives and suggestions.

The members of the *Open Systems Lab* over the years, Mark Astley, Svend Frolund, Nadeem Jamali, Wooyoung Kim, Rajendra Panwar, Anna Patterson, Shangping Ren, Dan Sturman, Prasanna Thati, Abhay Vardhan, Carlos Varela, James Waldby, Joonkyoo Yoo, Reza Zaiei and others have provided many useful comments during the course of this work. Thank you all for making my years at OSL so memorable. I would also like to thank Bonnie Howard for all her affection and help – my sojourn at OSL would not have been the same without her.

I wish to acknowledge my fellow scientists and managers at Hewlett-Packard Laboratories, where I spent many memorable years, for their support in helping me complete my disserta-

tion. My collaboration with Srinivas Ramanathan was instrumental in developing some of the techniques for multimedia load balancing presented in this thesis. Working with members of the Media Server Research Team – Manu Thapar, Vivian Shen, Weiping Lu, Glenna Mayo and Shenze Chen, has been a delightful experience. Rajiv Gupta, Arindam Banerji, Alan Karp, Rich Friedrich, Gita Gopal have encouraged me and helped reinforce my research vision. In addition, I wish to thank my friends, too numerous to name individually, for providing the much needed diversions from work and school and for their support and friendship. I would like to thank my dance and music teachers for bringing rhythm and harmony into my life.

My sincerest thanks to my family who have encouraged me through the years and have provided incredible support through good and hard times. I sorely miss my late grandmother, V. V. Lakshmy, who always kept my spirits up by her kindness and childlike innocence. My uncles, aunts, cousins, brothers-in-law and sisters-in-law, nieces and nephews have all in their own way helped make life a pleasant journey for me. Special thanks to my parents-in-law, Krishna Shankar and Pushpa Mehrotra for their kind words, affection and encouragement. A warm hug to my little dog, Tinku, who always brings joy to my heart.

I am deeply indebted to the special person in my life, my husband, Sharad Mehrotra. His affection, advice, encouragement and great sense of humor have enriched my life. From spirited research discussions to stuffed animals, I thank him for his unconditional support in all my endeavors. His energy and zeal have motivated me to do my best. I will cherish his love and the life we have together forever.

Finally, all the efforts leading to this dissertation would have been impossible without the sacrifice, efforts and guidance of two special people, my parents, who made it the purpose of their lives to help me live my dreams. My late father, Prof. V.S. Venkatasubramanian, I thank for inspiring me by his innate humility, brilliant mind and his eternal spiritual guidance. I am grateful to him for leaving with me a legacy of love for learning and peace. My mother, Saraswathy, has been my pillar of strength all these years. She has taught me to believe in myself, to never give up, and has been an example to me through her dedication, faith and hard-work. She has given up everything to be here for me. To her, I owe this dissertation.

Table of Contents

Chapter

List of Figures

List of Tables

Chapter 1

Motivation

In the coming years, multimedia to the desktop and home is likely to become a pervasive technology. Distributed multimedia(MM) applications are likely to become ubiquitous and influence the way computer systems are used and developed. In recent years, advances in networking, internetworking, storage and hardware technologies are making it possible for intensive applications like real-time interactive multimedia to the home and desktop a reality. With the advent of global applications like the Web and interactive community services, distributed information access is having an impact on the masses. Interconnectivity and distribution of services and information is becoming widespread. For example, future clinical environments and medical information systems will require the storage, retrieval, navigation and presentation of multifaceted information with stringent requirements on the reliability and accuracy of data. The concept of multimedia is also becoming essential in future educational and entertainment systems, as evidenced by the development of digital studios and instructional video-on-demand technology. Multimedia objects in the above systems are stored in high capacity storage devices and deliver interactive, digital MM services over emerging residential and enterprise broadband networks. These networks support hundreds of thousands of subscribers – leading to the merging of computing and communication systems.

This new generation of systems is required to service requests with widely varying characteristics - from broadcast/multicast services to on-demand/interactive services; from static,

non-continuous media like text and images to more dynamic continuous media types like video. Multimedia applications are often time-constrained – perceptual semantics of the information is dependent on the timeliness of arrival.These requirements are stated as end-to-end Quality of Service (QoS) specifications and imply real-time information extraction, delivery and presentation.

Since these systems are commercially deployed, performance and cost-effectiveness play an important role in system design. For instance, one of the major design considerations for a large-scale system is scalability from both application and system perspectives - i.e. the ability to admit and service thousands of user requests simultaneously and the ability to add and remove nodes from the system dynamically. With enhancing demands for better cost-performance, scalability, and availability, merely pumping additional hardware will be of little use without effective management of the system resources. New algorithms, protocols and architectures for the end-to-end management of system resources like data, computation and communication in distributed multimedia systems must be developed.

A wide range of protocols and activities must be composed to implement end-to-end distributed application management. These protocols and activities must execute concurrently, non-disruptively and share the same resources. In order to avoid resource conflicts, deadlocks, inconsistencies and incorrect execution semantics, the underlying resource management system must ensure that the simultaneous system activities compose in a correct manner. The difficulty in reasoning about system level interactions is due to the complexity of characterizing the semantics of shared resources and specifying what correctness of the overall system means. In addition, the presence of user-specified QoS criteria that may need to be satisfied further complicates the allocation and management of resources.

In this thesis, we propose some resource management techniques that can be employed to guarantee cost-effective QoS in distributed multimedia environments. We address the issues of correctness and performance in the design of these systems. This thesis is divided into 3 sections: (1) Core services for distributed resource management (2) Composition of distributed

resource management activities and formal correctness reasoning (3) Providing QoS guarantees in distributed multimedia systems through composite resource management.

1.1 Core Services for Distributed Resource Management

Systems that exhibit a high degree of dynamicity and autonomy can be characterized as open distributed systems (ODS). ODS evolve dynamically and components of ODS interact with an environment that is not under their control. Very often, these systems are federated with multiple control domains. Adaptive resource management strategies must allow for dynamic adaptation of applications to new service parameters as resource availability varies and gracefully degrade the quality of service in the event of a failure.

The abstractions most natural for representing ODSs are inherent in the framework of distributed objects or actors. The actor model of computation has a built-in notion of encapsulation and interaction and can be viewed as a model of coordination between autonomous interacting components. In order to comprehend the interactions between system and application activities and between system level activities themselves, we distinguish between application activities and system activities. and reason that correctness criteria are met. In addition, we can use the basic policies to satisfy QoS criteria like dependability and predictability and reason that correctness criteria of system level activities like task scheduling and event management are met in the presence of these constraints.

While designing distributed resource management algorithms, correctness and consistency of the underlying system and executing applications must be ensured. To manage the complexity of reasoning about components of ODS, our strategy is to identify key basic services provided by the system where non-trivial interactions occur. We refer to these key services as *core services*. The runtime activities and primitives based on these core services can be used as a foundation for the implementation of other resource management activities and protocols. In this thesis, we describe and formulate core resource management mechanisms that can be used for the management of objects in a widely distributed system.

The core services we discuss in this thesis are remote creation, distributed snapshots and name services. Remote creation can be used as the basis for designing algorithms for activities such as migration, replication and load balancing. Distributed snapshots are used as the basis for global activities like distributed garbage collection, checkpointing and recovery. Part of the difficulty with automatic garbage collection in systems of *active* objects, such as actors, is that an active object may not be garbage if it has references to other reachable objects, even when no other object has references to it. This is because an actor may at some point communicate its mail address to a reachable object thereby making itself reachable. Because messages may be pending in the network, the asynchrony of distributed networks makes it difficult to determine the current topology. Using the core distributed snaphot service, we describe a generation based distributed garbage collection algorithm which does not require ongoing computation to be halted during garbage collection. This makes it possible for the system to provide real-time response to service requests. Similarly, we illustrate Similarly, a naming service can be used to design access control mechanisms, routing policies and group based communication. Resources in a distributed system can be partitioned into local and global resources. Many activities like memory management, scheduling etc. have both local and global counterparts. The partitioning of resource management responsibilities between these components is critical; local optimization of resources does not necessarily imply that the global system is efficient.

1.2 Composing Resource Management Activities

In order to be able to provide customizable and adaptable execution of concurrent services, ODS must provide support for the correct composition of these dynamic system services. Reasoning about the composition of distributed algorithms involves reasoning about the correctness (safety and liveness) of each algorithm and reasoning about the compositionality (non-interference) of multiple algorithms. We establish a framework for specifying concepts and stating requirements in an ODS. By defining a formal semantics for resource management, we establish a basis for specifying and reasoning about properties of and interactions between components of

such systems. The model we propose is the basis for developing a semantic framework that supports dynamic customizability and separation of concerns in designing and reasoning about components of open distributed systems.

In our view, a system is composed of two distinct kinds of objects or actors - application-level and system-level objects distributed over a network of processing nodes. Application-level objects carry out application level computation, while system objects are part of the runtime system that manages system resources and controls the runtime behavior of the application level. Conflicts and interference can arise between application-level objects, between application and system levels and between the system level objects themselves. Thus, standard safety and liveness properties are not adequate to specify components of ODS. Non-interference properties must also be specified and checked. Making non-interference properties explicit is a means of making specifications modular and composable.

We express the core services defined in the previous section in this two level model and specify more complex services in terms of purely system-level interactions, which are better understood, although still non-trivial. We prove properties of non-interference and composition of resource management activities in this framework. We also develop a formal foundation for dynamic installation of resource management activities and protocols in an executing system. This, we believe is a step towards development of a powerful tool for designing practical and robust operating environments for open distributed systems.

1.3 Distributed Multimedia Systems - An Application

In this part of the thesis, we use the developed theoretical framework to specify and manage the requirements of multimedia systems. Advances in technology are facilitating the spread of widely distributed multimedia services. These high-performance systems are being deployed over wide-area networks to deliver a variety of interactive, digital multimedia services to residential subscribers [**?**] and enterprise systems. Distributed multimedia architectures meet the scalability and geographic distribution requirements in such large deployments [**?**].

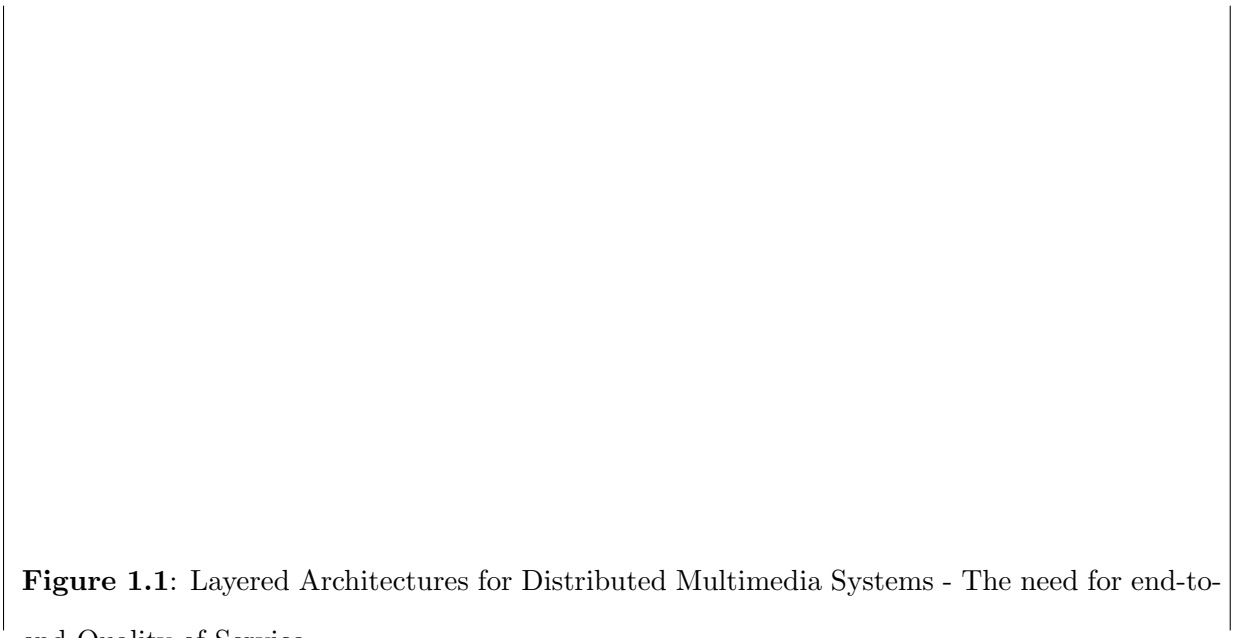

Figure 1.1: Layered Architectures for Distributed Multimedia Systems - The need for end-to-end Quality of Service.

Although service requirements for many applications are stringent, most MM applications can tolerate minor, infrequent violations of their performance requirements. This degree of freedom is specified as a quality-of-service (QoS) parameter, for e.g., quality of delivered image, tolerable jitter in a video frame, and must be enforced by the DMM system. There are two main phases associated with providing predictable quality of service to every incoming request - static and dynamic. The more static aspects of QoS management are handled by negotiation, resource reservation and admission control protocols executed when a service is initially established. QoS enforcement deals with the runtime monitoring, control and adaptation of services and resources to ensure that the desired QoS levels are sustained. For guaranteed QoS, this process must be applied to all components on the transmission and computational path - storage system, transmitting host, gateways and networks to the destination endpoint. The layered architecture in Figure 1.1 depicts the distribution of QoS components in the system.

DMM systems employ a wide range of protocols - communication, transport, media-encoding, negotiation, resource reservation protocols that must be composed for end-to-end transfer. The simultaneous execution of multiple management policies at different points in the stream require

system designers to guarantee correct and safe compositions of protocols and system activities. The cost-effective realization of layered protocols implies efficient resource utilization while composing protocols.

In this section of the dissertion, we discuss facilities required for dynamic adaptation of global multimedia applications to varying service parameters and system conditions. In our approach, we visualize MM applications as a collection of autonomous, concurrent information processing entities called media-actors. We study linguistic constructs for the synchronization of multiple media-actors and propose a meta-architectural framework for QoS-based resource management of media-actors. We address two important issues:

- **Specifying Quality-of-Service (QoS):** Global networks we know today such as the Internet, are best-effort traffic systems with data services that provide access to mainly non-continuous media. However, multimedia applications are characterized by the presence of QoS requirements for video quality, tolerable jitter, delays etc. Modular specification of QoS requirements helps in reasoning about system interactions in the presence of QoS constraints. This in turn, helps us deal with the complexity of widely distributed MM services.

- **Cost-effective resource management:** Better performance and availability require additional system resources, e.g.,processing power, network and storage resources. Cost-effective utilization of system resources is also needed to accommodate the rise in the number of service requests. Customizable system architectures help in managing the end-to-end performance while guaranteeing desired QoS to the application. This in turn requires the development of new algorithms, protocols and architectures for the management of data, computation and communication.

1.4 Thesis Organization

The remainder of this document is organized as follows. Chapter 2 sketches examples of core resource management services and develops algorithms for management of actors. We specify

core resource management services – remote creation, distributed snapshot and naming services that can be used as a basis for more complex activities. We illustrate how these core services may be used in implementing more complex activities within the actor framework, e,g, migration and distributed garbage collection. Similarly, we illustrate how migration and replication services can be defined using the core remote creation service.

In Chapter 3, we present a two-level model of distributed computation based on actors. that provides for the separation of application and system requirements. We describe the TLAM (the Two Level Actor Model), the formal model for specifying and reasoning about distributed RM activities, its components and semantics.

In Chapter 4, we sketch examples of specifications and compositions of services: remote creation, migration, and reachability snapshots and indicate how the specifications might be used to build running systems. We discuss the specification of core services in the two-level metaarchitecture - remote creation and reachability snapshot at different levels of abstraction and formally show how an implementation of these services satisfies the required service specifications. We also reason about how these services can be safely composed to execute simultaneously by defining the interactions between base and meta-actors and meta-actors themselves. We discuss invariants that must be satisfied in order for the migration and reachability snapshot services to co-exist.

In Chapter we explore some of the policies, mechanisms and compositions required to specify and manage distributed multimedia systems. We address the following issues: (1) Deciding what constitutes quality of service metrics - balance between user-satisfaction and effective resource management. (2) Specification of QoS in the actor model (3) Defining a multimedia management architecture using the two level meta-architectural approach and performance evaluation. We investigate metric parameters for video QoS based on conflicting requirements from two perspectives: the user who desires improved quality and the service provider who desires effective system utilization. Based on results from empirical studies, we define parameters of resource consumption (storage and network bandwidth etc.) and user satisfaction (jitter, synchronization skew) and derive analytical interrelationships among the metric parameters. We

also describe challenges in the modeling and specification of timing related multimedia (MM) services in open distributed systems. We propose a specification of timing related Quality-of-Service (QoS) attributes in an actor-based distributed system using a real-time variant of actors known as *RtSynchronizers* and describe some techniques for informally reasoning about quantitative QoS properties.

In Chapter 6,we describe a two-level multimedia management architecture for composite resource management in distributed MM servers. We formulate various policies for load management in distributed video servers using a two-level architecture - (a) replication, placement and migration of video objects using the core remote creation service, and (b) dereplication of video objects using annotations obtained by a snapshot of the current executing state of the system. For scheduling requests, we propose an adaptive scheduling policy that compares the relative utilization of resources in a video server to determine an assignment of requests to replicas. We also propose a predictive placement policy that determines the degree of replication necessary for popular objects using a cost-based optimization procedure based on a priori predictions of expected subscriber requests. To optimize storage utilization, we also devise methods for dereplication of video objects based on changes in their popularities and in server usage patterns. We simulate an object-based multimedia VOD system that consists of application video objects servicing video streams from multiple servers and a metalevel load-management system that manipulates application execution to improve performance. We show how application objects can be managed effectively by composing multiple resource management activities managed at the metalevel. Performance evaluations indicate that a load management procedure which uses a judicious combination of the different policies performs best for most server configurations.

In Chapter 7, we conclude with a summary of the dissertation and future research directions.

Chapter 2

Core Services for Distributed Resource Management

There are some features that distinguish a distributed, decentralized system from a centralized system. They include:

- Lack of knowledge of global state
- Lack of a global timeframe
- Nondeterminism

Distributed algorithms have been proposed to provide various distributed system services: routing, deadlock-free packet switching, global traversal, election, termination detection,global snapshots and fault tolerance. In order to understand the implications of concurrency in these system services, we must first be able to specify and understand interactions involved in each service. We can then talk about composing multiple system services and reason about overall system state. In this chapter we describe three core services – remote creation; directory services and identity; and snapshot services on which multiple algorithms can be based.

2.1 A Model of Concurrency - Actors

Distributed systems of the future must handle applications that are open and interactive, where problems that need to be solved can dynamically change. Hence such systems must be reactive, open to change and be able to deal with inputs that vary in volume and kind. The system must evolve dynamically based on inputs from an outside world and exploit concurrency efficiently under different circumstances. To realize the potential and limitations of distributed systems and services, we need an effective abstract model to reason about them. An abstract model facilitates the development of various applications without any concern for the underlying architectural configuration. It is also possible to insulate hardware and software developments, and derive the benefits of advances in one without having to be overly concerned about equivalent advances in the other.

One can view a distributed computation model as a set of abstractions that capture the semantics and functionality of concurrent program execution. Many models of concurrency and distributed computing have been proposed [**?**, **?**, **?**, **?**, **?**].

With MM transmission, there is a rising demand for an infrastructure that supports a wide range of services and applications. With global multimedia applications, the main challenge is that of openness and performance in a distributed environment. Systems exhibit continuous evolution and change in topology with dynamically varying request patterns, information, users and services. Dynamic introduction of real-time services and communication is critical to manage this complexity. What is required is a model that allows distributed applications to be developed, implemented and enhanced while maintaining the desired level of service quality. We characterize such systems as open distributed systems (ODS). The *Actor model* or *Actors*, first proposed by Carl Hewitt [**?**] and later developed by Agha [**?**], captures the essence of concurrent computation in open distributed systems at an abstract level.

2.1.1 About Actors

Open Distributed Systems (ODS) evolve dynamically and components of ODS interact with an environment that is not under their control. The Actor model of computation has a built-in notion of encapsulation and interaction, and thus it is a natural model to use as a basis for a theory of ODS.

In the actor paradigm, the universe contains computational agents called *actors*, which are distributed in time and space. Traditional passive objects encapsulate state and a set of procedures that manipulate the state; actors extend this by encapsulating a thread of control as well. Each actor potentially executes in parallel with other actors and may send messages to actors it knows the addresses of. The Actor model of computation has the following characteristics:

- Each actor has a conceptual location (its *mail address*) and a *behavior* as illustrated in Figure 2.1. Actor addresses may be communicated in messages, allowing dynamic interconnection governed by *locality laws*.

- The communication topology of an actor system is called the *acquaintance relation*. An actor can only send messages to its acquaintances. The acquaintances of an actor are among those it is given at creation time, those it has created and those sent to it in a message. The acquaintances that an actor can be given at creation time are among the acquaintances of its creator. An actor can also forget acquaintances. Thus the topology can change dynamically.

- Finally, new actors may be created; such actors have their own unique addresses. On receiving a communication, an actor processes the message and as a result may cause one or more of the following events:

    1. Creation of a new actor,
    2. Change of behavior, and
    3. Sending of a message to an existing actor.

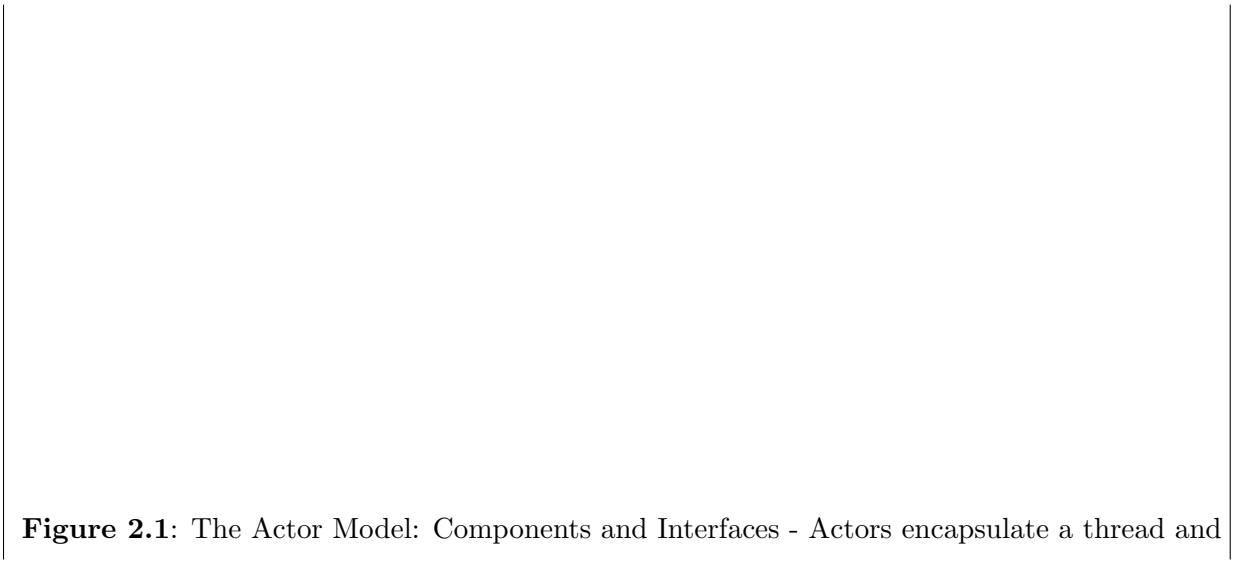

Figure 2.1: The Actor Model: Components and Interfaces - Actors encapsulate a thread and state. The interface is comprised of public methods which operate on the state.

In general, *Actors* can be viewed as a model of coordination between autonomous interacting components. The local computation carried out by the components may be specified in any sequential language.

Axioms expressing the essential features of actor computation such as the acquaintance relation and ordering of events are given in [**?**]. Will Clinger [**?**] developed a powerdomain semantics of actor systems, showing the consistency of these axioms. An interleaving transition system semantics for an actor language is given in [**?**, **?**]. This work builds on the formulation in [**?**] and develops methods for reasoning about equivalence of actor programs [**?**, **?**] and the composition of activities in actor systems [**?**]. (See [**?**, **?**, **?**] for more discussion of the Actor model, and for many examples of programming with actors.)

Note that the Actor model is, like the theory of higher order nets or the π-calculus, general and inherently parallel. A key difference between actors and objects defined using ports in the π-calculus is that actor names (addresses) are uniquely tied to the identity of an actor – giving out an actor name does not enable the recipient to receive messages directed to that actor. Moreover, asynchronous communication in actors directly preserves the available potential for parallel activity: an actor sending a message does not have to necessarily block until the re-

cipient is ready to receive (or process) a message. Of course, it is possible to define actor-like buffered, asynchronous communication in terms of synchronous communication, provided dynamic actor (or process) creation is allowed. On the other hand, more complex communication patterns, such as remote procedure calls, can also be expressed as a series of asynchronous messages [**?**]. Note also that Actors can be used to express different forms of concurrency and parallelism. Data parallelism is expressed as a broadcast message sent to multiple actors. Functional parallelism is expressed by concurrent messages to multiple actors whose responses can be synchronized later via join continuations [**?**].

2.1.2 Evaluation of the Actor Model

Portability of Actor Programs

A concrete way to think of actors is that they represent an abstraction over concurrent architectures. Actor primitive operations provide a simple and powerful base on which to build higher level features and abstractions for concurrent programming. These include programming abstractions for specifications of key concepts such as communication, synchronization, scheduling and placement [**?**]. An actor runtime system provides the interface to services such as global addressing, memory management, fair scheduling, and communication. It turns out that these services can be efficiently implemented, thus raising the level of abstraction while reducing the size and complexity of code on concurrent architectures [**?**]. Application portability is achieved via a well-defined interface exported to the compiler – this makes the application architecture independent. Architecture dependent modules are separated; hence porting the runtime system across multiple platforms is fairly straightforward. Actor languages can be implemented on a number of computer architectures such as sequential processors, shared memory processors and SIMD architectures. However, multicomputers are particularly interesting because of their scalability characteristics. Actor languages have proven especially useful as a language model for computation on multicomputers [**?**] because the implementation of actors on message-passing architectures is straight-forward. The network in multicomputers supports the actor mail ab-

straction; memory is distributed and information is localized on each computer. Load balancing and managing communication patterns are simplified by other characteristics of actor systems such as the use of small objects which can be created and destroyed dynamically. It should be observed that actors can be directly supported on multicomputers whereas implementing other paradigms on such computers may require their implementation in terms of some simple variant of the actor execution model [**?**].

Scalability and Openness

The Actor model promotes scalable computing via the use of suitable programming abstractions. Group abstractions allow us to simplify computation and enhance performance in applications where there is uniformity of behavior among a group of actors. Hence, data parallelism available in an algorithm can be expressed concisely and naturally. ActorSpaces is an extension of the actor paradigm with Linda-like primitives for group based communication. The ActorSpace model allows the abstract specification of a group of actors with specific attributes. The sender of a message specifies a destination pattern which is pattern matched against the attributes of the actors in the actorspace. Concurrent Aggregates (CA) is an extension of the Actor model which defines multi-access aggregates that allow several messages to arrive simultaneously, removing the need for serialization of messages. Concert is a system based on CA [**?**], that facilitates the expression of irregular parallel programs and the construction of large applications. Metaarchitectural frameworks using the Actor model [**?**] provide dynamic customizability in designing and reasoning about components and protocols in open concurrent systems.

Performance Issues

Effective implementations of distributed systems involves tradeoffs in expressiveness, complexity and efficiency. With distributed programs, one cannot separate algorithm and program design from architectural details. For instance, the partitioning and distribution of data can dictate which of the potentially parallel operations may be executed sequentially. Hence, we

cannot determine actual performance and cost metrics of a distributed algorithm independent of architecture and other system policies. The Actor model provides scalability and portability. However, since *Actors* is largely a programming model, and the representation of architectural issues is abstracted, it is hard to directly model performance parameters using actor specifications alone. Since actors may vary in granularity and the cost of communication may vary significantly, it is impossible to determine the cost of computation of many actor programs in any abstract way.

Moreover, because the Actor model is highly distributed, compilers must serialize execution to achieve execution efficiency on conventional processors. The efficiency of a distributed algorithm implemented via actors is dependent both on problem decomposition into actors, as well as the placement of actors in a specific architecture. One of the most effective compiler transformations is to eliminate creation of some types of actors and to change messages sent to actors on the same processor into function calls.

Actors can also be used to provide efficient portability by using the variable grain size feature. The programmer can optimize the size of the sequential process within an actor to match the optimal grain size of processes on a given parallel architecture based on costs associated with process creation, context switching and communication. Hence, we can obtain efficient implementations by code transformations of actor programs to actors of suitable granularity, e.g. by creating *fatter* actors that throttle concurrency.

Overall suitability of Actors

The specification, modeling and efficient programming of distributed applications is key to expanding the influence of distributed computation. Actors can provide a suitable execution environment for distributed programs. Specifically, in order to determine the balance between efficiency and flexibility, we need to utilize compilation techniques that help minimize performance penalties due to abstraction. Problems exist in developing programming paradigms with abilities to describe varied interaction patterns. For example, the information provided by a transitional model of actor systems is too detailed to be used for reasoning about complex

system structures. Ensuring consistency and correctness without inhibiting concurrency is a difficult challenge. Thus, there is a need to develop a language for building abstractions which includes a methodology for combining modules. This requires the development of a calculus of components and configurations. Specifically, it should be possible to directly model and reason about abstractions rather than analyzing the actor implementations of those abstractions. This is an important topic of ongoing research.

2.2 Core Services for Open Distributed Systems

Open distributed systems should provide strong support for customization and adaptation. Traditional reflective systems aim at providing a customizable and adaptable execution of concurrent systems [**?**, **?**, **?**]. However, these systems are very difficult and complex to reason about. Non-reflective systems which support customization do so only on a static basis. In an object-oriented system such as Choices [**?**], or Spring [**?**], frameworks may be customized for a particular application. However, once customized, the characteristics may not change dynamically.

What we require is an architecture that allows runtime aspects of an application and interaction protocols to be programmed and tuned independently of the basic application behavior. This independence allows a wide variety of protocols and aspects of execution behavior to be composed from a small number of basic modules. An example of this modular composition is the *layered* or onion-skin approach for combining dependability protocols [**?**]. Another essential characteristic of an open system is the ability to install system and runtime services on the fly. This gives us the ability to add new features to an operating system without *halting* the system or *rebooting* – an essential characteristic of open systems.

Distributed systems in practice employ a large number of ad-hoc mechanisms and policies to provide performance and reliability. Services such as load balancing, replication and migration involve the recreation of a service or data object at a remote site. Services such as distributed garbage collection, debugging, synchronization, checkpointing, global traversal, termination

detection and consistency protocols require the recording or capture of information on multiple nodes and links, i.e., information obtained by a snapshot of some aspect of the node or link. Services like access control, deadlock-free routing, group-based communication, distributed data structures, etc. require interaction with a global *repository* or nametable. In this chapter, we consider three core services that represent this classification: remote creation; directory services and identity; and snapshot services. We define core services as those basic system services where complex interactions between the system and application can occur. These core services form the building blocks for other classes of algorithms that rely on the core services for complex system interactions. Our goal is to be able to reason about the interactions between the layered services in terms of interactions among the core services. We use the commonly observed patterns in distributed algorithms to identify three basic activities:

- Recreation of services/data at a remote site
- Capturing information at multiple nodes/sites
- Interactions with a global repository.

Correspondingly, we define three core services - remote creation, distributed snapshot and directory services. The organization of the core services and their use in distributed resource management activities is illustrated in Figure 2.2.

Another aspect of this work is to develop systematic means of describing and ensuring constraints on interactions among system-level activities. Typically, while reasoning about resource management activities, we tend to assume independence of resource management activities from each other. But, in a real system resource management activities are not necessarily independent. In addition, interleaving the application-level and system-level execution may give rise to inconsistencies if improperly done. For example, complications may arise in a system that facilitates migration if global snapshots can be taken concurrently. In the case of distributed GC, migration of actors would interfere with the recording of a consistent GC acquaintance relationship. By implementing the interaction interface at a very basic level, i.e. in the core

Figure 2.2: Classification of Core Services

services we can reason about and detect safe points in a dynamic system where system activities can be performed.

Two simple forms of combinations of the core service specifications are:

- *Using the core services:* In this chapter, we demonstrate how system level activities in a distributed system like migration, replication, security, distributed garbage collection etc.can be implemented using these core services as a foundation.

- *Co-existence of multiple resource management services:* By forcing layered services to obey certain restrictions specified as constraints or invariants, we achieve safe co-existence of multiple policies. To illustrate combination by co-existence we identify conditions to permit Migration and Reachability Snapshot services to co-exist (act concurrently) in a system without interfering with one another.

2.3 The Remote Creation Core Service

Remote creation is the process by which actor creation occurs on a specified node other than the node from which creation is being initiated. Remote creation is a basic facility that can be used in other resource management activities like load-balancing, replication and migration.

Figure 2.3: A Remote Creation Request.

By encapsulating the interactions between the application and system level actors within the remote creation service, we can state requirements that ensure safe and correct composition of other resource management activities with remote creation. Consequently, services using the remote creation facility are ensured of correct interactions without interference from other system activities.

A remote creation request has 2 components – *ad* a description of the fragment to be migrated and a node, N (See Figure 2.3). This is interpreted as a request to create an application level fragment consisting of the actors created and messages sent when *ad* is executed on node, N, in some configuration. The fragment is independent of the node and configuration up to choice of new actor addresses. No acknowledgement is required for a remote creation request. If the requester needs to know if the request has been met, or names of some of the newly created actors, then this can be arranged by specifying appropriate messages as part of the requested fragment, and observing their delivery. This technique is illustrated in the Migration and Service Replication Behavior specification described in the following subsections.

Compiler transformations can be used to enhance the performance of remote creation. Effective compiler transformation techniques have been suggested to eliminate creation of some

Figure 2.4: Using the Remote Creation Core Service - to build replication and migration services.

types of actors and to change messages sent to actors on the same processor into function calls. For instance, the time taken for remote actor creation varies considerably as it is dependent on processor load and network traffic. Hence, in platforms where hardware context switching is available, it is desirable to implement a technique called *split-phase remote creation* [**?**]. Here, the processor requesting remote creation context switches to another thread from the one requesting the remote creation effectively hiding latency and improving processor utilization. This approach is however not suitable on platforms where context switch is expensive. Another approach to hide remote creation latency is to define an alias or local clone for the actor requesting remote creation. The alias returns a handle representing the newly created remote actor to the actor requesting remote creation which then proceeds with the rest of its computation. The alias then handles the remote creation process independently. The significant cost difference between local and remote message sends presents another opportunity for optimization. Assigning higher priority to processing local messages will simplify message queue management and reduce the cost of message scheduling.

In the following section, we describe in detail two uses of the remote creation core service to provide higher level facilities – actor migration and service replication (as illustrated in Figure 2.4).

2.3.1 Migration - Using the Remote Creation Core Service

Migration is the process by which actors and their associated address spaces move from one node to another. The migration service allows for relocation of actors for easier access, availability and load balancing. In this section, we first provide the specification of a generalized migration service. We then give a behavioral specification of the migration service using the remote creation service specification of the previous section. We also expose some high level invariants that one class of implementations of the service might rely upon. This also provides a basis for expressing properties of interaction of the migration system with other system level services.

2.3.1.1 Migration Service

A migration request is given by a pair (α, ν) where $\alpha \in Act_{\mathtt{b}}$ is the actor to be migrated, and ν the destination node. This is interpreted as a request to move the computation carried out by α to the node ν. In order to state explicitly invariants maintained by the system during the migration process, we classify the migration process into 3 phases wrt the actor being migrated and the node to which it is being migrated. The first phase, C_0, is the initiation phase and specifies the state of the system when the migration request received can be processed. It determines the computation to be migrated by suspending the computation of the actor and noting its current description. In the second configuration, C_1, the actual actor migration is performed. The final configuration, C_2, finalizes the migration process and establishes transparent access to the migrated actor. The system progresses from C_0 to C_1 to C_2.

2.3.1.2 Migration Behavior based on Remote Creation Service

Migration behavior is specified by assigning to each node a migration system on that node that handles migration requests for actors on that node. A remote creation service accessed via a remote creation request *rcRequest* is used to install the migrating actors state on the remote node. The remote creation request also includes a message to be sent to the original node

Figure 2.5: Stages of Migration

containing the address of the newly created actor. To avoid confusion with other messages to the migrating actor, α, a temporary actor α_s is created to receive this message.

The specification of the migration behavior based on the remote creation service refines the 3 stages of the service specification. When the migration system receives a request to migrate an actor, $MigRequest(\alpha, \nu)$, the following actions are executed by the migration mechanism. The three ovals in Figure 2.5 depict the three stages of migration.

- **Initiation Phase:**

  1. Create a surrogate actor α_s on the original node to receive the newly created actor address. The sole job of this surrogate is to receive the new address of the migrated actor.
  2. Replace the behavior of α by a queue.
  3. The remote creation service registers to be notified when the surrogate actor α_s receives a message.

4. Send a remote create message to the desired node with the description of the actor configuration to be created and the surrogate actor address to which the remote address must be sent.

- **Remote Creation Phase:** The remote creation is executed and α' is created with the desired behavior and α' is sent to α_s.

- **Rerouting or Finalization Phase:** The address of the newly created component is delivered to the surrogate and this signals the migration service to complete migration. The behavior of the original actor is changed to that of a forwarder, which forwards pending and incoming message for α to α'.

2.3.2 Replication - Using the Remote Creation Core Service

Replication is an essential facility in distributed systems for multiple reasons -

- to reduce network traffic
- reduce latency of access and retrieval of remotely located information
- to increase availability and dependability of information.

In this section, we consider replication for dependability.

Dependability or the ability to maintain availability of system facilities in spite of software or hardware failures is one of the important services in a distributed system. In this section, we represent a composable, application-independent mechanism to guard against fail-stop failures in a system by using replication. The replication protocol modeled here has been described in [**?**], which introduces MAUD (Meta Architecture for Ultra Dependability). Based on the degree of consistency desired by the application, replication can use one or more of the core services. Firstly, replication can be implemented as state capture followed by remote creation on a replica node using the captured state. Secondly, if distributed consistency constraints are to be met, the remote creation phase can be preceded by a global snapshot that records a

consistent state specified by the activity. Global snapshots are discussed in more detail in the following section.

A server is an actor system with a single receptionist that expects a request for the service along with a reply address. For the present we will assume that a server is not history sensitive (has no internal state). To increase dependability, a service can be replicated. To do this, several exact copies (replicas) of this service are created on different processing nodes in the system. Any request to the service is rerouted to all the replicas by a *distributor* and responses are intercepted by a *collector*. The first response received for a client (application actor's) request is returned and further responses from other replicas are ignored. In this way, there is a high probability that the client process will receive a response to its request even if one (or more) of the nodes fail. A replicated server is again a server (exactly one reply is sent in response to each request), and replication does not change the functionality of a service.

One way of modelling dependability via replication in the two-level model is to define a dependability subsystem in which there is a Replication Dependability Actor (RDA) on each node. To replicate a service, a request is sent to an RDA containing the address of the service actor to be replicated, and a list of nodes on which it is to be replicated. When such a request is received, the RDA creates a collector actor C, and sends a request to the RDA actors on the nodes in the list to create local replicas of the service using collector C. When the addresses of the requested local replicas have all been sent to the initiating RDA, a distributor behavior is created and installed in place of the service behavior. The behaviors of the distributor, collector, and replicas are is described below. Figure 2.6 gives a picture of a replicated server.

- **Distributor:** The Distributor (called serveMQ in [**?**]) has as acquaintances the list of replicas. It also has a generator of new identifiers. When a client request arrives, a new identifier is generated and a message containing the identifier and the original request is sent to each replica.

- **Collector:** The collector (called serveDispatcher in [**?**]) maintains a log of replies received for each request, using the unique identifier associated with each request. When

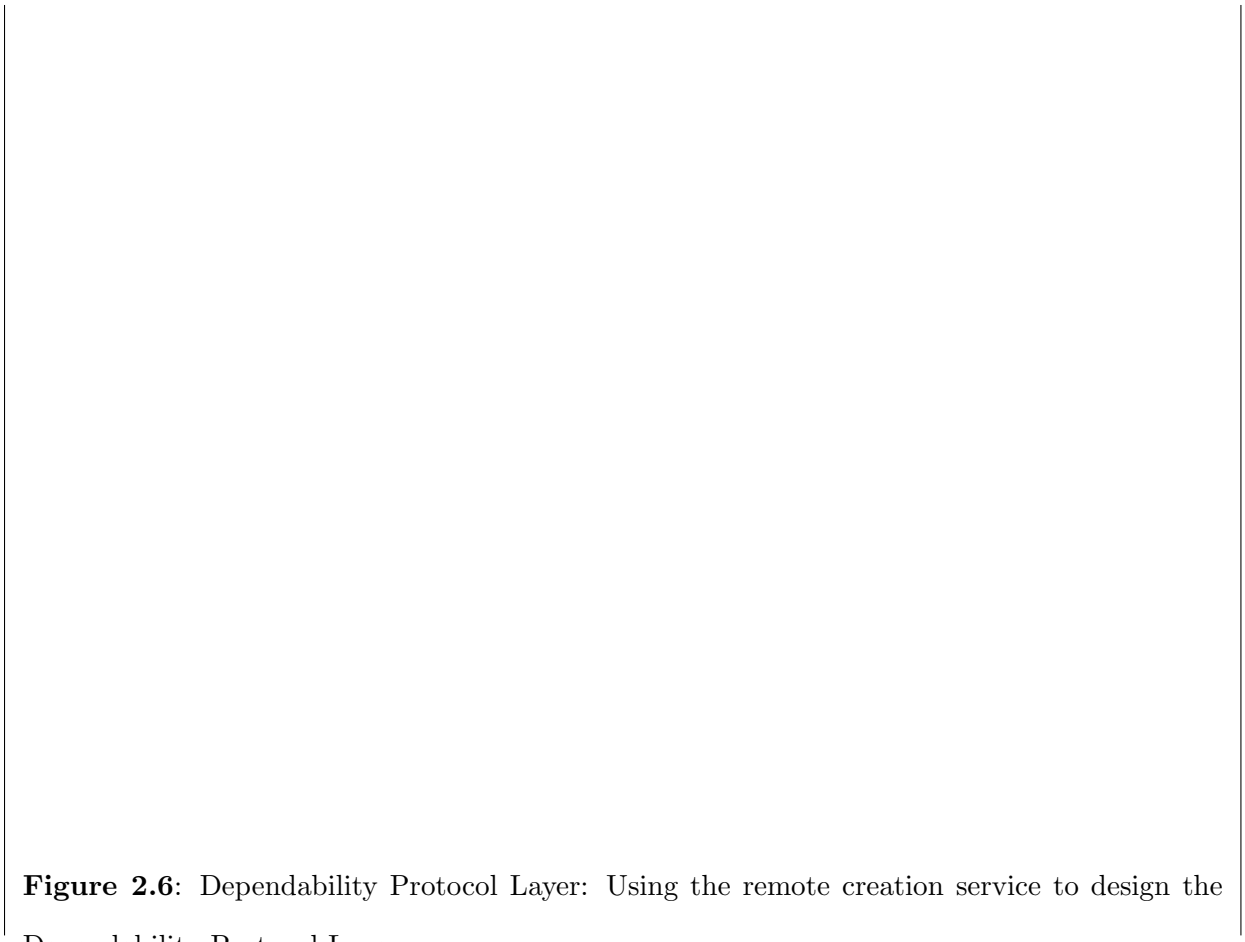

Figure 2.6: Dependability Protocol Layer: Using the remote creation service to design the Dependability Protocol Layer

a reply with identifier *id* is received from a replica, the collector looks to see if there is a previous reply to the corresponding request. If not, the identifier and reply are logged and the reply is sent on to the customer. Otherwise the reply is logged and discarded. Each entry thus contains the number of replica responses received. When all replicas have sent a response, the entry can be deleted from the response table.

- **Replica:** A replica consists of three actors: an actor with the same behavior as the original server; a receiver; and a transmitter, that appears to the outside as a single actor whose address is that of the receiver. The receiver receives messages containing an identifier and a client request. When the server is ready for the next request, the receiver sends the identifier and client address to the transmitter and sends the request with the client's address replaced by the transmitter address to the server actor. The receiver then queues further requests from clients until the transmitter signals that the server has replied to the current request. When the transmitter receives a reply, it signals the receiver and sends a message containing the identifier, the reply, and the client address to the collector. Thus the receiver/transmitter behave as pre/post processors of messages to the server that make the request identifier transparent to the server.

2.4 The Distributed Snapshot Core Service

A generalization of the distributed snapshot mechanism can be used for a variety of applications where some global aspect of the distributed system needs to be observed. For example, global information about an actor subsystem is useful for distributed debugging or checkpointing a distributed system. Global properties like the number of application-actors, number of messages being processed and task queue sizes help in making runtime decisions like load balancing and migration leading to efficient runtime management of a distributed system. To fully represent the global state of the distributed system, we need a mechanism for recording the state of all nodes including the portion of node state being communicated in the network channels [**?**]. As state information is accessible explicitly only in nodes, a snapshot mechanism must ensure

that node state information in channels are recorded at some node in the system (possibly the target node itself). In order to initiate snapshot recording on every node and force messages in channels to reach a node, we need to be able to define protocols for message propagation giving rise to the following kinds of waves:

- A wave that visits all nodes exactly once broadcasting specific information.
- A wave that traverses all links in the system exactly once forcing messages on channels to reach nodes (where their state can be recorded) in the direction of its flow. This wave may also propagate information to nodes.

The starting and finishing points of both waves, the propagation path and constraints on message propagation must be well defined. Furthermore, termination must be signaled when the wave is complete, i.e. when all nodes in the system have been visited or when all links in the system have been traversed. This termination signal will also serve as a synchronization point in the snapshot mechanism. Note that this generalizes the notions of wave and global snapshot discussed in [**?**] where the snapshots explicitly do not account for information contained in messages in transit.

We assume that the nodes in the network topology are ordered, such that it is possible to designate a *start* node and a *finish* node. We also assume message order preservation (FIFO assumption) over a single link, but not necessarily between any two nodes. The multi-cast messages used are of two kinds: *broadcast messages* and *bulldoze messages*. There are two types of bulldoze messages, *forward* and *backward*; every node has a set of forward bulldoze neighbors and a set of backward bulldoze neighbors. *Forward bulldoze messages* are initiated at the start node and propagate from every node to its respective forward neighbors. *Backward bulldoze messages* are initiated at the finish node and propagate from every node to its respective backward neighbors. Bulldoze messages traverse every pair of links in the network and, by the FIFO assumption on links, force messages already in the network to be cleared along the direction of the bulldoze. The propagation of a bulldoze message in a two-dimensional grid forms a wave as illustrated in Figure 2.7.

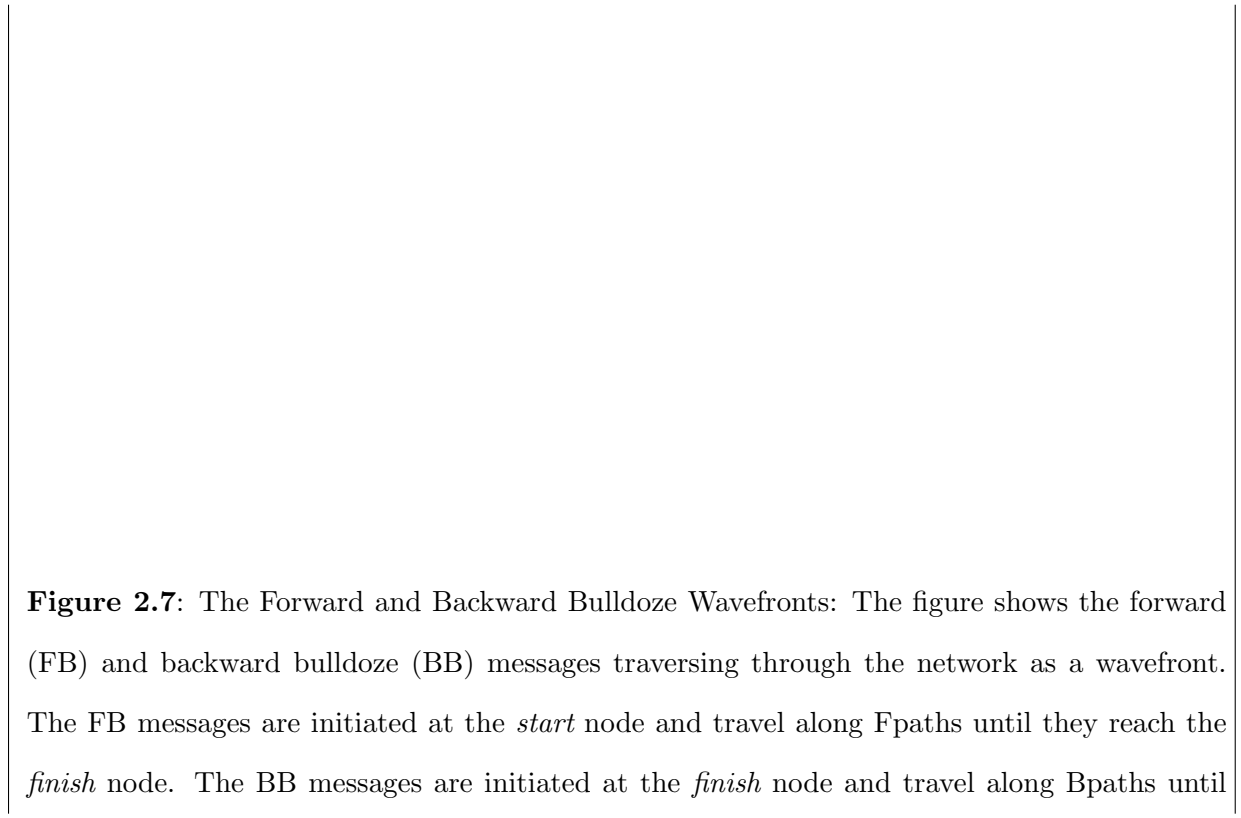

Figure 2.7: The Forward and Backward Bulldoze Wavefronts: The figure shows the forward (FB) and backward bulldoze (BB) messages traversing through the network as a wavefront. The FB messages are initiated at the *start* node and travel along Fpaths until they reach the *finish* node. The BB messages are initiated at the *finish* node and travel along Bpaths until they reach the *start* node.

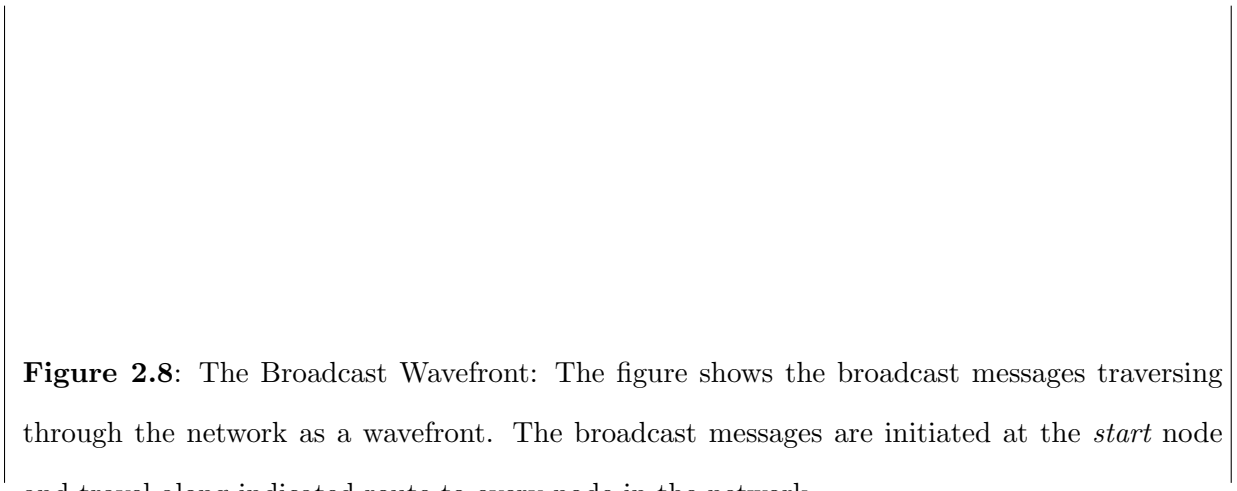

Figure 2.8: The Broadcast Wavefront: The figure shows the broadcast messages traversing through the network as a wavefront. The broadcast messages are initiated at the *start* node and travel along indicated route to every node in the network.

In the case of broadcast messages, every node has a set of broadcast neighbors. *Broadcast messages* are initiated at the start node and propagate from every node to its respective broadcast neighbors. Broadcast messages reach every node in the system, but traverse only a subset of the links and hence are less costly. The protocol for propagating a broadcast message in a two-dimensional grid is illustrated in Figure 2.8.

The broadcast and bulldoze messages can be used along with appropriate actions to record the necessary information in a distributed system. In the distributed GC algorithm, for example, the snapshot waves were used to record a global acquaintance relation. Information collected at each node may be processed in two ways:

- Locally recorded information is processed locally with further passes.
- Locally recorded information is reported to a centralized *collection service* that processes the snapshot information.

A global snapshot root-actor in the entire system coordinates system wide actions like the switching on/off of snapshots, initiation of the propagation waves and detection of termination. It serves as the interface to any service that requests a snapshot. In addition, every node in the system has a snapshot meta-actor that processes snapshot waves and records requested

local information pertaining to the snapshot. In addition to the snapshot of the global state on every node, one may require global information on the state of the channels, e.g. message routing information, congestion along channels, network latencies, link failures etc. used to make runtime decisions. This may be done via special monitors initiated to record channel information.

Characterizing global properties of asynchronous distributed computation models is a difficult task. In the next chapter, we define a two level architecture using which we can determine specifiable global properties of a system to make a number of runtime decisions. Note that not all global information can be observed or recorded reliably: for example, global clocks and network channel states. The representations we propose are useful in determining reasonable approximations of these properties and using these approximations to prove related safety and liveness properties of a distributed system.

2.4.1 Distributed Garbage Collection - Using the Snapshot Service

In this section, we illustrate how the snapshot service may be used to form the basis for a global activity, distributed garbage collection. We present a non-halting garbage collection algorithm for active objects in a distributed system [**?**, **?**]. We describe a mechanism called HDGC (hierarchical distributed garbage collection), suitable for systems of active objects distributed across a network of nodes. Hierarchical organization provides for scalability. It partitions a distributed system into smaller subsystems, which in turn may be further partitioned. The topmost level of the hierarchy is the entire system and the lowermost level of the hierarchy has a single node per subsystem.

An important advantage of our algorithm is that it is non-disruptive: it does not halt or otherwise interfere with the ongoing computation process. A novel feature is the recording of a GC-snapshot to obtain a consistent local and global view of the accessibility relation. The algorithm is described in terms of the actor model. However, it is applicable to any language supporting dynamic creation and reconfiguration of objects (passive or active), executed on a

network with a *global name space* distributed across the nodes [1]. The HDGC algorithm can be adapted to a wide range of parallel architectures including fine, medium or large grained MIMD machines, message passing, shared memory or distributed shared memory machines, or networks of workstations. In this section, we present the conceptual aspects of the algorithm.

2.4.1.1 Notion of reachability in actor-based systems

Reachability is a fundamental concept in actor systems. It characterizes the potential for communication of one actor with another. It is also the basis for memory management and other resource management activities. In this section we give a definition of reachability that takes into account the ability of an active object to become known by communicating its mail address. Here we are concerned only with reachability of application level actors. We then present an algorithm for marking objects that are reachable according to this definition.

In the actor model, the actor-level communication topology changes dynamically. Actor identifiers may be communicated to an actor, and alternately, an actor may change its behavior and lose an acquaintance. Furthermore, the ability of an "apparently unreachable" object to send its mail address to a reachable object must also be considered. Our definition of reachable in an actor-based system is derived from the work of Kafura et. al. [**?**]. The *root set* is a fixed set of actors from which reachability is traced. At a given point in time, the *acquaintances* of an actor are those actors whose address is 'known' by that actor, i.e. addresses occurring in the behavior of the actor or in messages sent to the actor but not yet processed (undelivered messages). It includes actors referenced in the current computation state of the system (environment variables, control structures like stacks etc.). Similarly, an actor is considered *busy* or *enabled* in a configuration if the busy status of its state indicates it is busy, or if there is an undelivered message to that actor (since delivery sets the busy status to true).

In an actor computation, the transitive closure of the acquaintance relation starting from the root set is not adequate to determine reachability. This is due to the fact that in systems of active objects, such as actors, an apparently unreachable object may at some point commu-

[1] This memory architecture is often referred to as *distributed shared memory*

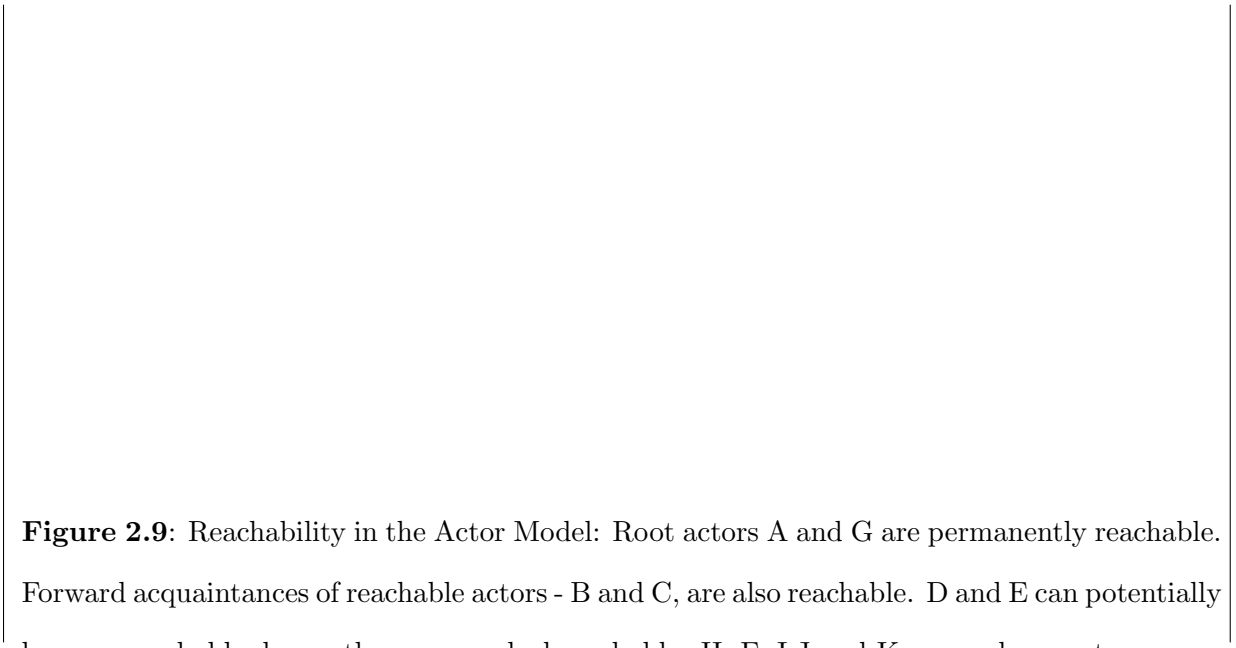

Figure 2.9: Reachability in the Actor Model: Root actors A and G are permanently reachable. Forward acquaintances of reachable actors - B and C, are also reachable. D and E can potentially become reachable, hence they are marked reachable. H, F, I,J and K are garbage actors.

nicate its mail address to a reachable acquaintance thereby becoming reachable. Thus inverse acquaintances must also be considered in determining reachability. The *inverse-acquaintances* of an actor in a configuration are the actors that have that actor as an acquaintance in the configuration. An actor is considered *inactive* in a configuration if it is not busy. It is *permanently inactive* if it is inactive, and it is not connected to a busy actor via some chain of inverse acquaintances. Figure 2.9 illustrates the notion of reachability in the presence of acquaintance relationships, busy actors and root actors.

Using these informal notions, the set of *reachable actors* is defined inductively as the least set such that:

- A root actor is a reachable actor.
- Every forward-acquaintance of a reachable actor is reachable.
- If an actor is reachable, then every inverse acquaintance of that actor which is not permanently inactive is reachable.

An actor that is not reachable according to the above definition is *garbage* and resources allocated to this actor can be reclaimed.

A GC snapshot of the system state determines a conservative approximation of the acquaintance relation. Two important properties of reachability for one level actor systems that make this a safe criterion for garbage collection are: an unreachable actor never becomes reachable; and the description of an unreachable actor can be replaced by a null behavior without changing the observable behavior of the system.

2.4.1.2 Obtaining a Consistent Reachability Snapshot

A GC snapshot consists of acquaintance and active status information that determines a consistent global view of the state of the system at start-of-GC time. Each node records, for each of its actors, its GC-acquaintances, its GC-inverse-acquaintances, and whether or not it was active at start-of-GC time. The GC-acquaintances of an actor are the current acquaintances, plus any acquaintances in messages in the network prior to the start of actual garbage collection. This is a safe approximation of the actors acquaintances, and insures that actors actually forgotten by one actor but sent in messages during GC will not be lost. The GC-inverse-acquaintances of an actor the set of actors having that actor as a GC-acquaintance. This information is used to account for apparently unreachable actors that might communicate their mail addresses to a reachable actor. The GC acquaintance information is used only for GC and can be discarded when the GC for which it was created is complete. For a global snapshot of the state of the system, we need to guarantee that both *local consistency* and *global consistency* have been achieved. Every node in the system needs a point of reference in time with respect to which it determines the accessibility or inaccessibility of actors in its memory. Once a node has established this point and recorded the necessary information, we have attained *local consistency*. *Global consistency* is a point in time when all participating nodes have agreed on a particular state of the distributed system.

In order to determine which messages were in the network prior to the start of GC and which entered after, ordinary messages are given *tags* to classify them as *old* or *new* messages.

Old (resp. *new*) messages are messages which were created prior to (resp. after) the time of the GC snapshot. When GC is initiated, all messages in the network are tagged *old*. During the process of recording the GC snapshot, the network will be cleared of *old* messages by means of the forward and backward bulldoze messages explained above.

To obtain the GC snapshot, first a pre-GC message is broadcast to every node in the system. When a node receives the pre-GC broadcast message, it initializes the GC-acquaintances of each actor residing on that node with (1) its current acquaintances and (2) all acquaintances contained in messages currently residing in its mail queue. Any acquaintances contained in *old* messages subsequently obtained from the network are added to the GC-acquaintances. It also initializes GC-inverse-acquaintances to be empty. When the pre-GC broadcast is complete, a pre-GC Fbulldoze message is initiated (by the finish-node that receives the final pre-GC broadcast message). When the pre-GC Fbulldoze message passes a node, it marks as active any objects with non-empty mailqueue. The active status of this node is retained for the current GC even though the node may become inactive during GC. Any messages subsequently communicated from that node are tagged *new*. The *new* tag on a message guarantees the recipient of the message that any acquaintances communicated in the message have already been accounted for. When the Fbulldoze message reaches the finish node a Bbulldoze message is initiated. When the Bbulldoze message passes a node, this signals that the recording of GC-acquaintances is complete. The node sends *I-know-you* messages from each of its actors to each GC-acquaintance of that actor. When an I-know-you message from actor A to actor B is received then actor A is added to the GC-inverse-acquaintances of actor B. A second forward and backward bulldoze phase is required to clear the network of I-know-you messages. This is initiated by the start node upon completion of the first backward bulldoze wave. When the second forward/backward bulldoze wave is complete, the start node sends a pre-GC-complete message to the root node. At this point, all *old* and *I-know-you* messages in the system have been cleared from the network and the snapshot information is recorded.

The backward bulldoze messages are needed for both the recording of GC-acquaintances and GC-inverse-acquaintances, since the forward bulldoze only clears forwards links and there may

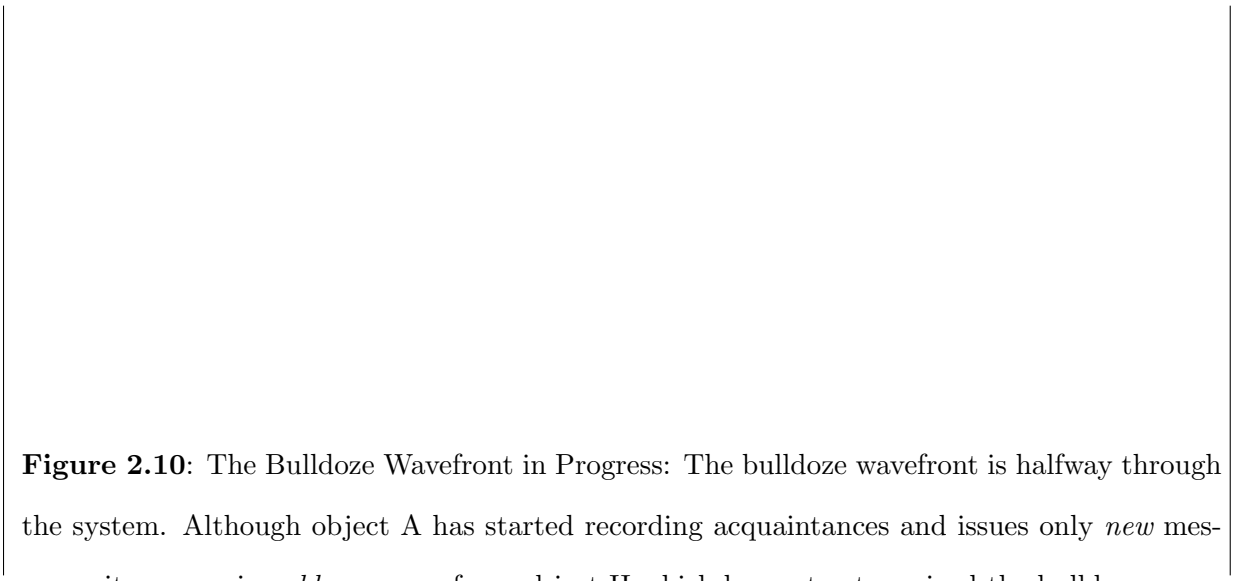

Figure 2.10: The Bulldoze Wavefront in Progress: The bulldoze wavefront is halfway through the system. Although object A has started recording acquaintances and issues only *new* messages, it can receive *old* messages from object H which has not yet received the bulldoze wave.

be messages traversing backwards links that need to be recorded. To see this, note that after an object, say A, has received the pre-GC Fbulldoze message it can send only *new* messages. However, it may receive *old* messages from an actor H which has not yet received the pre-GC Fbulldoze message (see Figure 2.10).

2.4.1.3 Using the Reachability Snapshot to Achieve Distributed Garbage Collection

Step 1: Pre-GC. In a system with distributed state there is no uniquely determined global state. Thus to compute some property of the state it is generally necessary to determine a global snapshot that determines a consistent view of the state. In the case of the acquaintance relation for an actor system, the problem of obtaining a consistent global snapshot involves an additional subtilty. The asynchrony of communication together with the ability to communicate acquaintances means that at any given time, there can be communications in the network whose acquaintances are no longer acquaintances of the sender, and not yet acquaintances of the receiver. This means that before a snapshot of the acquaintance relation can be taken, the network must be cleared of such communications. During the pre-GC step each node is notified

that a GC has been initiated, and the network is cleared of messages in transit at the time GC was initiate. This defines a local start-of-GC time on each node that is globally consistent. Each node records GC information relative to its start-of-GC time that will persist throughout the duration of the GC. The combined local information forms a consistent global snapshot of the system state that is adequate to determine the reachability of each actor in the system. We call this the *GC snapshot*. A detailed description of the information recorded and the process of recording the GC snapshot is presented in Chapter 5.

Step 2: The Distributed Scavenge Phase. During this step, actors that are non-garbage relative to the GC snapshot are marked *touched*. We use the definition of reachability in actors to determine non-garbage and develop a *distributed scavenge algorithm* for marking non-garbage actors [**?**].

Step 3: Local-Clear Initiation. Each node in the system is informed that the distributed scavenge phase has completed and local clearance begins. On each node, objects not marked touched are cleared from local memory, according to a local memory management on a node, and any other actions (updating receptionist tables, etc.) entailed by this reclamation are carried out.

Step 4: Local-Clear Phase. This step detects when all nodes have completed the local clearance initiated in the previous step.

Step 5: Post GC Broadcasts. This step informs each node that the current GC is complete: each node can now note that GC is no longer in progress and update necessary information to reflect this state. At the end of this step a new GC can be initiated at anytime.

2.5 The Directory Core Service

Name services in distributed systems provide a mechanism for identifying objects and locating them. Using a flexible naming scheme offers support for transparent heterogeneity where objects are transparent to the details of the underlying architecture. In the ActorSpace model [**?**], extensions to naming schemes provide flexible coordination patterns. Similarly, with naming as a basis, we can define complex interaction patterns and security mechanisms. In the following section, we discuss one extension of a naming facility – access control.

2.5.1 Security and Authorization - Using the Directory Core Service

In this section, we discuss the representation of a protection mechanism that maintains the integrity of information represented or stored within the system. Enforcement of security constraints is critical in a distributed system, where multiple users have the ability to access and manipulate remote information. Note that system security is built upon the assumption of system dependability, i.e underlying hardware failures may jeopardize the integrity of the system.

2.5.1.1 A model of security

We assume an actor(object)-centric model of security. An actor holds the power to grant specific authorities to other actors. This defines who can access the actor and in what way. We will use *capabilities* as the mechanism used to implement access control. A *capability* is a representation of the access control policy in a system. Access to objects is based on the ownership and presentation of capabilities. As the ownership of a capability controls access to actors in a system, creation of capabilities must be localized. Capabilities are created on behalf of the accessee actor and then communicated to other actors using a specified policy. A capability has two components (see Fig 2.11): (1) a unique actor address that identifies the actor on which the access is attempted and (2) an encoding of the access rights allowed by an actor holding this capability. The actor specified within a capability is the owner of the capability.

There are three operations that may be performed on capabilities:

- Granting capabilities
- Verifying capabilities
- Revoking capabilities

Granting a capability is initiated by the actor that owns the capability. The owner of a capability communicates the capability to one or more of its acquaintances (who can send a message to the actor). If capabilities are designed to be first class objects, they may be communicated in messages to the acquaintances of an actor along with actor addresses. Every message in the system is tagged with a capability. Access verification is performed dynamically when the message is processed at its target. If the capability on the message is validated by the receiver, the message is processed, else it is rejected.

In an actor based system with first-class capabilities, revocation of capabilities introduces further complications. In this scenario, it is possible for an actor to receive a revoked capability from one of its inverse acquaintances. Say an actor A is the owner of a capability Cap. A grants the capability Cap to its acquaintances B and C who communicate Cap in turn to their acquaintances. If A revokes capability Cap from B, there is no guarantee that B will not re-receive Cap from another of its inverse acquaintances at some later point in time.

However, if capabilities are not first class objects, the security mechanism may be tailored to allow revocation of capabilities. A restricted security mechanism would suffice for applications like database systems with a small class of operations (update and select). The choice of the capability mechanism therefore depends on the tradeoff between flexibility and control desired.

2.5.1.2 Representing access control using the directory service

We introduce the notion of a security actor that is responsible for access control decisions of a base-actor or a group of application-actors. Implementation of an exclusive security actor per application-actor is both expensive and unnecessary for most applications. Let us assume that there is a single security actor per node responsible for authenticating access to all application actors on that node.

Let us first consider the representation of capabilities as first class objects in the actor system. Here, the programmer of the actor system allocates capabilities and passes them around. The maintenance of system integrity is the responsibility of the programmer who is allowed to hand capabilities to base-actors. In this case the security actor is only responsible for verifying that messages to an actor on that node have the appropriate capabilities. Outgoing messages may be tagged with capabilities, if required, by the user.

Alternately, the capability mechanism may be maintained strictly as a system-level feature. In this case, capabilities given to an actor are recorded by the security actor on that node. Capabilities are only granted and revoked and they may not be communicated from an actor to its acquaintances. Insertion of capabilities into the outgoing message is done exclusively by the security controller (meta-actor) on that node. This causes some limitations in the actor model where mail addresses may be communicated dynamically. A mechanism must now be implemented to define access rights to the new inverse acquaintance of an actor or allow delegation of authorities. As before, an incoming message is tagged with a capability and authentication checks are performed before dispatching the message to its destination base-actor. The security controller on a node has 2 tasks to perform:

- It verifies that messages to an actor on the node have necessary capabilities
- It maintains capabilities associated with every object on the node in a *capability table* and tags outgoing messages with appropriate capabilities.

Figure 2.11 illustrates the *capability table* on a node that maps the actors on a node to the capabilities that they possess. Any outgoing message is tagged with an appropriate capability by the security actor on a node. Similarly, an incoming message is validated by the security actor before it reaches the actor that processes the message via notifications.

Using this basic security facility, it is possible to implement a variety of authentication policies. For example, a group based access policy may be enforced within the security actor that validates messages from users that belong to a certain group.

Figure 2.11: Security using Directory Services

Integration of system security with other runtime activities is critical with the increasing demand for *open* systems. Static authentication mechanisms are no longer sufficient to guarantee security in a dynamic and heterogeneous environment with varying protection requirements. The capability model described represents just one level of user security. An extended protection model would require the integration of multiple security mechanisms for ensuring network security and encryption facilities. Composition of different security schemes can be represented in the meta-architectural framework (described in the next chapter), by layering the security subsystem, with each layer implementing different granularities of security. Furthermore, the motivation for combining different runtime policies to generate dependable, secure systems requires us to reason about fault-tolerance and authorization in a unified framework and analyze their interaction.

2.6 Composing Services

A wide range of protocols and activities must be composed to implement end-to-end distributed application management. These protocols and activities must execute concurrently, non-disruptively and share the same resources. Problems with distribution and composition

include resource conflicts, deadlocks, inconsistencies and divergence resulting in incorrect execution semantics. The underlying resource management system must ensure that the simultaneous system activities compose in a correct manner. The difficulty in reasoning about system level interaction is due to the complexity of characterizing the semantics of shared resources and specifying what correctness of the overall system means. In addition, the presence of user-specified QoS criteria that may need to be satisfied further complicates the allocation and management of resources.

Consider the following example. An interesting situation arises if migration is permitted in a distributed system. If a distributed snapshot is taken concurrently with migration, then the snapshot process may lose information, or possibly never terminate. Therefore, a request for a global snapshot must inform every node in the system of the request for a snapshot. All the nodes in the system must cooperate to (1) not initiate any new migrations (2) detect the completion of any occurring migration. The broadcast wave described earlier can be used to inform the nodes about the snapshot request and detect the completion of any ongoing migrations. Once migration has been turned off, the distributed snapshot is taken. All nodes in the system are subsequently notified of the snapshot termination and migration is turned on again.

Our long-term goal is to develop a library of core services and tools that will allow a wide variety of protocols and aspects of execution behavior to be composed in a simple manner. For instance, a number of languages and systems offer support for constructing fault tolerant systems. However most do not support the factorization of fault tolerance characteristics from the application specific code. Similarly developing systematic means of describing and ensuring synchronization constraints on interactions between system components can be complex. High level language constructs for expressing such constraints include *synchronizers* and *activators* [**?**]. By providing frameworks that permit resource management mechanisms and policies to be attached and detached dynamically, we allow the composition of multiple meta-activities such as check-pointing, fault tolerance protocols and synchronization without requiring that the representation of one mechanism knows about the other.

Chapter 3

MetaArchitectures for Distributed Resource Management

In this chapter, we present a two-level model of distributed computation based on actors. This model is the basis for developing a semantic framework that supports dynamic customizability and separation of concerns in designing and reasoning about components of open distributed systems (ODS). In particular, we would like to be able to consider separately issues such as: functional behavior of an application; failure semantics and fault tolerance protocols; and resource management issues such as memory management, load balancing, and scheduling.

3.1 The Two Level Approach

Open distributed systems should provide strong support for customization and adaptation. Non-reflective systems which support customization do so only on a static basis. In an object-oriented system such as Choices [**?**], or Spring [**?**], frameworks may be customized for a particular application. However, once customized, the characteristics may not change dynamically. Traditional reflective systems aim at providing customizable and adaptable execution of concurrent systems. For example, the scheduling problem of the Time Warp algorithm for parallel discrete event simulation is modeled by means of reflection in [**?**]. A reflective implementation of object

migration is reported in [?]. Reflection has been used in the Muse Operating System [?] for dynamically modifying the system behavior.

Reflection also underlies recent work in language and system design that supports customization and separation of design concerns [?, ?]. Representation of dependability protocols as meta-level programs is presented in [?]. Some of the more recent research on actors has focused on coordination structures and meta-architectures [?, ?, ?]. The Aspect Oriented Programming paradigm [?] makes it possible to express programs where design decisions(aspects) can be appropriately isolated permitting composition and re-use of the aspect code.

Many of these languages or systems lack clearly defined semantics. The TLAM (Two Level Actor Machine) model is a first step towards providing a formal semantics for such languages, and a basis for specifying and reasoning about properties of and interactions between components of such systems. A preliminary version of this model was presented in [?, ?]. In [?] the notion of *abstract actor structure* as a framework for high-level, programming language independent, specification of individual actor behaviors is developed. This is just the TLAM *two-level actor structure* notion restricted to purely base-level systems.

In the TLAM, a system is composed of two kinds of actors, base actors and meta actors, distributed over a network of processing nodes. Base level actors carry out application level computation, while meta-actors are part of the runtime system which manages system resources and controls the runtime behavior of the base level. The application level of the model refines the model of [?, ?], explicitly representing more of the runtime structures and resources. It also abstracts from the choice of a specific programming language or system architecture, providing a framework for reasoning about heterogeneous systems. Meta-actors communicate with each other via message passing as do base level actors, but they may also examine and modify the state of the base actors located on the same node.

The multi-model reflective framework (MMRF) for distributed object computation [?] has many of the same motivations and objectives as the TLAM framework. We briefly compare the two approaches. In MMRF an object is represented by multiple models allowing behavior to

be described at different levels of abstraction and from different points of view. In each model the behavior of an object is described by a meta object for that model and each meta object sees and acts on only one base level object. In the TLAM the actor representation is left quite abstract for the present. Each meta actor can examine and modify the behavior of a group of base level actors – namely those located on the same node. Some instances of the TLAM may have the many-to-one organization of MMRF, but that is not the only possibility. The more flexible relation seems appropriate when considering facilities that involve manipulation of collections of base level actors. Both MMRF and TLAM use reification (base object state as data at the meta object level) and reflection (modification of base object state by meta objects). Both have implicit invocation of meta objects in response to changes of base level state. This provides for debugging, monitoring, and other hooks. In the case of MMRF it is the only means of interaction between meta level objects associated to the same base object. MMRF provides for explicit invocation of meta objects by associated base objects. TLAM provides for full actor-style interaction of meta level objects, but not (yet) for invocation of meta objects by base objects. A language, AL-1/D, based on MMRF has been implemented however there has been no work on specifying a formal semantics. In the TLAM effort we have concentrated on defining a rigorous mathematical semantics and on using this semantics to develop concepts and methods for expressing and reasoning about properties of ODS and their components.

Because actors are history sensitive, there is potential for interference between actors with a common acquaintance. There is also the additional possibility for interference between base and meta level actors on the same node. A third possibility of interference is between meta actors implementing different services and thus modifying base level actors in possibly incompatible ways. Thus, standard safety and liveness properties are not adequate to specify components of ODS. Non-interference properties must also be specified and checked. Similar observations have been made for traditional one level systems. For example, in [**?**] Abadi and Lamport give a method for describing open components of concurrent systems using assumption/guarantee assertions [**?**]. Assumptions are requirements on the components environment. Here we consider

object based systems and have the additional important problem of ensuring that meta-level activities do not have unintended effects on the base level semantics. Note that making non-interference properties explicit is a means of making specifications modular and composable.

To manage the complexity of reasoning about components of ODS, our strategy is to identify key basic services provided by meta-actors where non-trivial base-meta interactions occur. With these basic services identified and specified, more complex services can be specified in terms of purely meta-level interactions, which are better understood, although still non-trivial. The basic idea follows principles of program development based on use of high level abstractions that hide much of the implementation complexity. Recent work applying these ideas to distributed computing includes [**?**] where programming tools, each with associated proof rules for reasoning about, them are proposed. What is new in our work is the application of these ideas to two level systems.

In this thesis, we consider two examples of basic services with non-trivial base-meta interactions: *remote creation* and *reachability snapshot*. The remote creation service allows a meta actor to effectively create a base level subsystem on a remote node. Remote creation can be used as the basis for services such as migration and replication in a distributed system. The reachability snapshot service provides the capability to record a snapshot of the base level reachability relation. This relation determines which actors can potentially be accessed and affect the observed system behavior. The TLAM model can be used to show that a reachability snapshot is a safe criterion to be used for garbage collection. Thus the reachability snapshot service can be used as the basis for design, implementation, and verification of a distributed garbage collection service (cf. [**?**, **?**]). The reachability snapshot specializes the notion of global snapshot of a distributed computation which is an important tool for distributed programming (cf. [**?**, **?**]). Reachability allows us to illustrate some of the issues that arise in reasoning about object based systems where object identity is a fundamental concept.

We have tried to specify useful services, however the point of the examples is not to propose standards, but to illustrate some of the kinds of properties, requirements and interactions that

we imagine will arise in practice and to take the first step in developing methods for specifying and reasoning about concurrent services. To illustrate the use of such specifications, we show how the remote creation service specifications can be used to develop and refine a migration service specification. To illustrate combination by co-existence we identify sufficient conditions to permit migration and reachability snapshot services to co-exist and act concurrently in a system. We also show how the non-interference requirements allow us to reason about composition of migration and reachability snapshot services.

Here we give two forms of specification for each service X. The first gives conditions under which a TLAM system is said to provide an X service, with respect to a function for creating request messages and possibly other parameters. The second gives conditions under which a TLAM system is said to have an X behavior with respect to a set of meta-actors and possibly additional parameters. The property of having an X behavior is linked to the property of providing an X service by defining a service request function in terms of the parameters of the behavior specification and showing that a system having X behavior provides the X service with respect to the defined request function.

Each form of specification itself may be refined in various ways. One form of refinement articulates non-interference requirements, still at a fairly abstract level. Such refinements generally correspond to design decisions related to the choice of a class of algorithms for implementing the service. Another level of specification is the notion of a group of actors providing a service. This moves from specification in terms of global system behavior to specifications of individual actors or collections of actors. Thus refinement of specifications provides another form of modularity and scalability, by reducing the task of implementation to that of implementing individual abstract behaviors.

Our various levels of abstraction include levels similar in spirit to the three abstraction levels identified in the OSI reference model and proposed to strengthen the enterprise level of the ODP Reference Model in [**?**]. These levels are: the combined behavior of a system and its

environment (service); the role of the system in this combined behavior (service provider); and the decomposition of this role (protocol).

Our work is unique in the use of the two-level framework as a formal semantic foundation for specifying and reasoning about applications, system level services, and their interactions in a common framework.

3.1.1 Other Related Work

A number of language independent formalisms have been developed for specifying and reasoning about concurrent systems. These include formalisms based on temporal logic [**?**, **?**, **?**], behavior histories [**?**], and I/O automata [**?**]. These formalisms provide a general framework for specifying safety and liveness properties and a means of organizing proofs. They have been used to specify and verify a variety of protocols. One difficulty with existing formalisms is that components are not represented as objects of the formalisms, rather as instances – signatures and formulae. Thus they do not address issues such as equivalence and transformations.

The Unity language [**?**] is a notation for describing systems, the focus of this work has been methods for program specification and development of proof rules that support reasoning about these specifications and their relations. The underlying semantic model is a state transition system. Unity is limited by the fact that there is no support for system decomposition or for talking about interactions of system components with one another or with the environment except via effects on shared global variables. Also since there is no distinction between system and application activities, programs must be transformed to implement additional services and protocols.

Our work on meta-architectures for ODS has been motivated by various work in the areas of programming language and system design, and distributed algorithms some of which is mentioned in the previous subsection. The work described in this dissertation has concentrated on developing a semantic framework and concepts for reasoning about ODS using ordinary

informal, but rigorous mathematics. Choosing or developing formal notations, proof systems, or logics is outside the scope of this work, although of great interest as a topic of future work.

In this chapter, we only give informal definitions of the structures and concepts introduced. A more rigorous specification and semantics for the TLAM that provides precise mathematical definitions is discussed in the appendix (See Appendix **??**). We try to use standard notation for sets, sequences and functions. Our notational conventions are described in detail in the appendix.

3.2 The TLAM Model - A Summary

In this section, we provide a brief summary of the TLAM model sufficient to understand the examples described in this thesis. A full description is given in the appendix.

3.2.1 TLAM Structure

A two-level actor machine (TLAM) consists of a two level actor structure (TLAS) distributed over a network of processor nodes and communication links. There is no shared global memory and no global clock.

A TLAS provides an abstract characterization of actor identity, state, communications, and computation, and of the connection between base and meta level computation. Base level actors and messages have associated runtime annotations that can be set and read by meta actors, but are invisible to base level computation. Actions which result in a change of base-level state are called events. In the TLAM, there are two kinds of events: delivery of a message to an actor, and local computation by an actor. In addition to changing the actors local state, local computation may result in creation of new actors, or sending of new messages. The TLAM event handling mechanism allows meta-actors to react to base-level events. This provides a flexible mechanism for interaction of meta-actors with the built-in runtime system. It also provides a clean way for specifying, reasoning about and designing daemons, monitors and similar facilities.

In more detail, a TLAS is a structure consisting of

- Sets:
  - Act, **Val**, **Msg**, Ad.
- Basic functions and relations:
  - $level$, ren, $busy$, acq, tgt, cnt,
  - $enabled_{\mathrm{del}}$, $deliver$, $getA$, $setA$; and
- Interpreters:
  - Ev_b, Ev_m, Ev_{eh}.

3.2.1.1 Sets

- Act is the set of actor identifiers. We let α range over Act (and by our meta-variable convention, see the appendix for details, α_0, α' also range over Act). Each actor in a system has a unique identifier, by which it can be known to other actors.
- **Val** is the set of values that may be communicated in messages. **Val** includes Act and we let v range over **Val**.
- **Msg** is the set of messages that actors may use to communicate with one another.
- Ad is the set of actor descriptions. An element ad of Ad describes the local state of an individual actor, and together with the interpreter functions, specifies the effects of an actor in this state executing a step. A description includes information traditionally contained in script or methods, contents of local or instance variables, and the local mail queue. In the case of meta actors, the description also specifies event handling actions.

3.2.1.2 Basic Functions and Relations

- $level$ partitions identifiers, values, messages, and descriptions into base and meta levels. We write $Act_{\mathtt{b}}$ for the set of base level identifiers and $Act_{\mathtt{m}}$ for the set of meta level identifiers. A similar convention is used for other partitioned sets.

- $ren(\rho, x)$ extends the bijection, ρ, on actor identifiers to descriptions, values, messages and other entities. This allows us to formalize the fact that the interpretation of actor descriptions is uniformly parameterized by the choice of identifiers for acquaintances. In particular, local creation of new actors can be described locally, by choosing locally new identifiers. Renaming is used to avoid name conflict when adding new actors to a system.

- $busy(x)$ returns true if the actor is currently *busy* and false otherwise. Being busy indicates that an actor is enabled for an execution step and could potentially send messages or create new actors without delivery of additional messages. For example, if the actor has not finished processing previously delivered messaged.

- $acq(x)$ is the (finite) set of acquaintances of (actor identifiers occurring in) a value or description x. The acquaintance function lifts homomorphically to structures built from actors, values and descriptions.

- $tgt(m)$ is the identifier of the target actor (receiver) of the message m.

- $cnt(m)$ is the contents (a communicable value) of the message m. We write $\mathbf{Msg}\alpha v$ for a meesage with target α and contents v.

- The test $enabled_{\mathrm{del}}(ad)$ determines if the actor description, ad, is enabled for delivery of a message.

- $deliver(ad, v)$ is the description resulting from delivery of a message with contents v to the description ad. Message delivery preserves level and after delivery a description is busy.

- Base level messages and descriptions may be annotated. Each annotation has a unique associated tag. Tags are just values that have no occurrences of actor identifiers, and annotation values are any value. $getA(ad, Tag)$ is the annotation of ad associated with tag Tag.

Figure 3.1: Interpreters - Base, Meta, Event. Actor configurations represent the combined state of a group of actors.

- $setA(ad, Tag, v)$ sets the annotation of ad associated with Tag to v. Message annotations can be similarly accessed and set.

3.2.1.3 Interpreters

To explain the interpreters we introduce the notion of actor configuration. An actor configuration represents the combined state of a group of actors, from some point of view. It is a finite, level preserving, map (notation ac) from identifiers to descriptions. A base level actor configuration is one whose domain consists of base level identifiers. Ev_b and Ev_m are the base and meta level interpreters, and Ev_{eh} is the event handling interpreter for meta level descriptions. Figure 3.1 indicates how the 3 different interpreters in the TLAM relate to each other.

- **Base Level Interpreter:** $Ev_b(\alpha : ad)$ specifies the effects of a (base level) actor α with local state ad executing a computation step. In particular it specifies: a description (not necessarily different) ad' of α after the step; a base level actor configuration, ac, (possibly

empty) describing the newly created actors; and a set, M, of messages sent. We say that upon execution *ad becomes* ad', *ad creates* actors, ac, and *sends* messages, M. The acquaintances of ad' and of the new actors and messages and the targets of the new messages are constrained to obey the locality laws discussed in Section 2.1.1.

- **Meta Level Interpreter:** $Ev_m(\alpha : ad)(ac)$ specifies the effects of (meta level) actor α with local state described by ad and access to base level actor configuration ac executing a step. In addition to specifying meta level effects analogous to those for base level execution, modifications to the base level actors in ac, and newly created base level actors and messages may be specified. A base level creator, possibly located on another node, must be specified for each newly created base level actor and a base level sender must be specified for each new base level message. This allows one to view the base level effects of a meta level transition as an extended form of base level transition.

- **Event Handling Interpreter:** $Ev_{eh}(\alpha{:}ad)(ac, event)$ specifies the effects of (meta level) actor α with local state described by ad and access to base level actor configuration ac in response to the event *event*. The meta level effects are the same as for execution steps. The only allowed base level effects are the modification of annotations of actors in ac. The event may be the delivery of a message to an actor in ac or the base level effects of an execution step. Meta-level transitions are constrained to obey two-level acquaintance laws (See Appendix **??**).

Some Simple TLAS Examples

Definition 1 (Sinks)

A base actor description, ad, is a *sink* if $deliver(ad, m)$ is a sink, and $Ev_b(\alpha, ad, new)$ returns a sink actor configuration, no changes, no new actors or messages, where $ac(\alpha)$ is a sink, for any base actor α, any appropriate allocation function new, and any (base level) message m.

Definition 2 (Forwarder)

Consider a base level system with a base level factory actor, f, whose state is described by

F and a meta level actor mf, whose state is described by MF. When a request from customer c is received by f, it creates a new actor, d, with initial description D and sends its name to c. $mf : MF$ reacts to the creation event by creating a new meta actor, say md, with description $MD[d, ?, 0, \emptyset]$. md creates a backup for $d : D$ and sends copies of all messages received by d to the backup. The ? is a placeholder for the name of the backup actor to be created. To preserve arrival order, the copies are paired with a number, initially), indicating the arrival rank at d of the message. Since md can not directly send messages to the backup when it is notified of a message delivery to d it must simply remember these and send the accumulated backup messages during its own execution steps. The final parameter of MD is the accumulated set of backup messages, initially empty. The interpreters for these actor descriptions are given by the following equations.

$$Ev_b(f : deliver(F, c))([d]) = ac[f, d, c]$$

$$\text{where} \quad ac[f, d, c] = f : F, d : D, \mathbf{Msg}cd$$

$$Ev_{eh}(mf : MF)(ac[f, d, c])([md]) = ac[f, d, c], mf : MF, md : MD[d, ?, 0, \emptyset]$$

$$Ev_m(md : MD[d, ?, j, X])([d']) = md : MD[d, d', j, X], d' : D'[0]$$

$$Ev_{eh}(md : MD[d, x, j, X))(\mathbf{Msg}dm) = md : MD[d, x, j + 1, X + <j, m>]$$

$$Ev_m(md : MD[d, d', j, X]) = md : MD[d, d', j, \emptyset], \mathbf{Msg}d'X$$

3.2.2 Network and Distribution

The network underlying a TLAM is a graph consisting of a set, **Node**, of processor nodes, and a set, **Link**, of directed links (one way communication channels) between nodes. The two level actor structure is distributed over the network by a function *loc* that maps actor identifiers to nodes.

This formulation of the TLAM structure emphasizes a systemwide view of configurations which is convenient for technical reasons. It is easy to give an equivalent formulation and distribution emphasizing local (per node) view.

3.2.3 TLAM Semantics

TLAM semantics is given by a labeled transition system, <*Cfig*, *Lab*, *Trans*>, which is determined by the TLAS and its distribution over the network. *Cfig* is the set of configurations (global states), *Lab* is a set of labels, and

Trans $\subset$ *Cfig* $\times$ *Lab* $\times$ *Cfig* is a labelled transition relation.

Configurations

A configuration, C, has three components. $lc(C)$ is a link configuration associating to each communication link a sequence of undelivered messages enroute along that link. $nq(C)$ is a node buffer configuration associating to each node a sequence of undelivered messages buffered at that node. The third component $ca(C)$ is the set of actor configuration maps. $ac(C)$ is an actor configuration – a level preserving finite map from identifiers to descriptions. If actor α is in the domain of this map, then $ac(C, \alpha)$ is the local state of α in the configuration. We write $Cast(C)$ for the domain of $ac(C)$ and we require that all actor identifiers occurring in C belong to $Cast(C)$.

Undelivered Messages

$Undel(C)$ is the set of undelivered messages in C, i.e. messages in the links or node mail buffers.

$$Undel(C) = \bigcup_{\gamma \in \mathbf{Link}} lc(C)(\gamma) \cup \bigcup_{\nu \in \mathbf{Node}} nq(C)(\nu)$$

$$Undel(C, \alpha) = \{m \in Undel(C) \;\Big|\; tgt(m) = \alpha\}$$

We extend functions on actor descriptions to configuration-actor pairs by first extracting the description of the actor from the configuration.

$$busy(C, \alpha) = busy(ac(C, \alpha)) \quad \text{if} \quad \alpha \in \mathrm{Dom}(ca(C))$$

$acq(C, \alpha) = acq(ac(C, \alpha))$ if $\alpha \in \mathrm{Dom}(ca(C))$

$getA(C, \alpha, Tag) = getA(ac(C, \alpha), Tag)$ if $\alpha \in \mathrm{Dom}(ca(C))$

$setA(C, \alpha, Tag, v) = setA(ac(C, \alpha), Tag, v)$ if $\alpha \in \mathrm{Dom}(ca(C))$

The restriction, $C/\mathtt{b}$, of a configuration, C, to the base level is given by the following:

- $lc(C/\mathtt{b})(\gamma)$ is the result of deleting all meta level messages from $lc(C)(\gamma)$;
- $nq(C/\mathtt{b})(\nu)$ is the result of deleting all meta level messages from $nq(C)(\nu)$; and
- $ac(C/\mathtt{b})$ is the restriction of $ac(C)$ to base level actors.

Transitions

There are two kinds of transitions: communication and execution.

- A communication transition with label $l = \mathtt{l2n}(\gamma, \nu)$ moves a message from a link γ to its target node ν. A communication transition with label $l = \mathtt{n2l}(\nu, \gamma)$ moves a message from a node ν to a connected link γ. A communication transition with label $l = \mathtt{del}(\nu)$ moves a message from a node ν to a target actor located on that node, if the actor is enabled for delivery, or to the end of the queue if the actor is not enabled for delivery.
- An execution transition with $l = \mathtt{exe}(()\alpha_F)$ is a computation step taken by a base or meta level actor, α_F, called the *focus actor*. The resulting configuration is obtained using the effects specified by the appropriate interpreter, using the focus actors current state, and in the case of a meta level transition, the configuration of base actors located on the focus actors node. Newly created actors are located on the focus actors node and newly sent messages are placed, in some order, at the end of the queue of undelivered messages of the focus actors node.

Events: A transition in which the base level system state on a node is modified (via delivery, creation, or sending) is called an event. After each event and before any other transition, each co-located meta level actor is notified and the configuration is further modified according to