Load Balancing of Autonomous Actors over Dynamic Networks

Carlos Varela, Kaoutar El Maghraoui, and Travis Desell

Department of Computer Science,

Rensselaer Polytechnic Institute, Troy, NY 12180, USA,

{cvarela,elmagk,deselt}@cs.rpi.edu,

http://www.cs.rpi.edu/wwc/

Load Balancing of Autonomous Actors over Dynamic Networks

No Author Given

No Institute Given

Abstract. The Internet is constantly growing as a ubiquitous platform for high-performance distributed computing. In this paper, we propose a new software framework for distributed computing over large scale dynamic and heterogeneous systems. Our framework wraps computation into autonomous actors, self organizing computing entities, which freely roam over the network to find their optimal target execution environments.

We introduce the architecture of our worldwide computing framework, which consists of an actor-oriented programming language (SALSA), a distributed run time environment (WWC), and a middleware infrastructure for autonomous reconfiguration and load balancing (IO). Load balancing is completely transparent to application programmers. The middleware triggers actor migration based on profiling resources in a completely decentralized manner. Our infrastructure also allows for the dynamic addition and removal of nodes from the computation, while continuously balancing the load given the changing resources.

To balance computational load, we introduce three variations of random work stealing: load-based (RS), actor topology-based (ARS), and network topology-based (NRS) random stealing. We evaluated RS and ARS with several actor interconnection topologies in a local area network. While RS performed worse than static round-robin (RR) actor placement, ARS outperformed both RS and RR in the sparse connectivity and hypercube connectivity tests, by a full order of magnitude.

1 Introduction

The constantly increasing performance of personal computers accompanied by rapidly growing network bandwidth has equipped the Internet with a large pool of abundant and inexpensive computational resources. A wide range of large scale, distributed and parallel scientific problems that require massive computation could benefit tremendously from this huge pool of inexpensive resources if we have the ability to find and utilize them efficiently and securely. Devising a scalable architecture to find these resources and distributing the computation efficiently among them is necessary to achieve worldwide execution of programs. Therefore, dynamically reconfigurable high-performance distributed middleware services need to be developed to address the challenges of dynamicity, heterogeneity, load balancing, fault tolerance, and security over highly dynamic open large scale computer networks.

With a large scale network, such as the Internet or the World-Wide Web, a single point of failure is unacceptable so a decentralized (a.k.a peer-to-peer) coordination model must be used. In a worldwide computing infrastructure, nodes can join and leave nondeterministically and applications must adapt to the changing environment. Manual load balancing and fault tolerance by application programmers in such large infrastructures is extremely time consuming, if not impossible. To address these complex tasks, we present three different strategies to autonomously load balance and tolerate faults in a dynamic environment.

The goal of our work is to create a smart, adaptive, dynamic and scalable computing middleware infrastructure where the computation of various scientific problems is distributed in a way transparent to the programmer. Our framework is based on the actor model [1] where computation and data are encapsulated within a single entity. Our extention to the actor model consists of *autonomous actors*, that are able to:

- Profile available resources
- Find and migrate to optimal execution environments, or *theaters*
- Replicate themselves to improve reliability
- Split and merge as more available resources emerge or disappear to improve performance

These operations are supported by middleware services forming part of a global Internet Operating System (IO) running on a single virtual worldwide supercomputer (the Internet). IO provides various services such as coordinating all the activities involved in distributing the computation among actors, managing the constantly changing resources (e.g. storage, memory, bandwidth, etc), and balancing the load among various nodes. There is a traditional tradeoff between information profiling and decision making. The load balancing strategies presented differ in the amount of information used to decide how to place the actors in the network and discuss their performance.

We provide preliminary empirical results using our own worldwide computing software framework. The worldwide computing framework consists of an actor-oriented programming language (SALSA) [2], a distributed program execution environment (WWC), and an internet operating system for autonomous distributed system reconfiguration (IO).

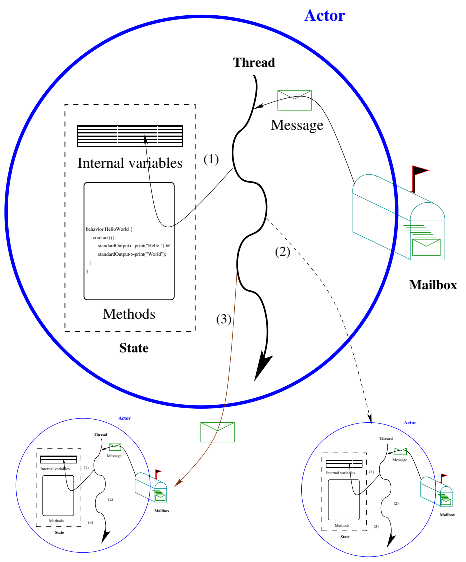
The rest of the paper is organized as follows. Section 2 introduces the worldwide computing framework and Section 3 discusses its implementation. In Section 4, we present the different load balancing strategies. Section 5 discusses our preliminary performance results. Section 6 discusses related approaches. Finally, Section 7 concludes our work with discussions and future work.

2 Model of Autonomous Worldwide Computing

2.1 Actors

The Actor model of computation is based around the concept of encapsulating state and process into a single entity. Each actor has a unique name, which can be used as a reference by other actors. Communication between actors is purely asynchronous. The actor model guarantees message delivery and fair scheduling of computation. Actors only process information in reaction to messages. While processing a message, an actor can carry out any of the three basic operations: alter its state, create new actors, or send messages to peer actors (see figure 1). Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [3] and facilitates mobility [4].

The actor model and languages provide a very useful framework for understanding and developing open distributed systems. For example, among other applications, actor systems have been used for enterprise integration [5], real-time programming [6], fault-tolerance [7], and distributed artificial intelligence [8].



```
behavior HelloWorld {
    void act(){
        standardOutput<-print("Hello ") @
        standardOutput<-print("World");
    }
}
```

Fig. 1. Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to peer actors

2.2 Universal Actors

In considering mobile computation, it becomes useful to not only model the interactions of actors between each other, but also to model the interactions of actors with their environments. In the actor model, locations are not explicitly represented, therefore semantically there is no difference if two actors are in the same memory space, or on two computers on opposite ends of the earth. However, when considering the problems associated with worldwide computing, it becomes important to represent the actor's environment, to account for different latencies or unreliable environments.

Universal actors are an extension to the actor model adding locations, mobility, and the concept of universal names and universal locators. Names represent actor references that do not change with actor migration. Locators represent references that enable communication with universal actors at a specific location. Each location represents an actor's run-time environment and serves as an encapsulation unit for local resources. Ubiquituous resources, such as CPU, have a generic representation – actors keep references which get updated to the new resources upon migration to a new location.

2.3 Autonomous Actors

When a system is composed of mobile actors, it can be reconfigured arbitrarily, as long as all its used resources are ubiquitous. Autonomous actors extend universal actors by:

– Profiling performance to improve quality of service.
– Migrating autonomously to balance the load.
– Spliting and merging to improve performance over dynamic networks
– Replication to tolerate failures.
– Introducing message priorities to allow for urgent reconfiguration messages.

Profiling Autonomous actors can adopt different profiling strategies, based on the way profiles will be used. Generally, actors profile processing power, memory, storage, latency and bandwidth. Each actor keeps a record of the number of messages received, messages sent, and messages processed. Autonomous actors can also profile where messages are sent to and received from, as well as the time taken to process or send a message. Based on the profiled information, the middleware's *decision agent* decides how the autonomous actors are to be distributed (More details about how the decision is made will be given in section 4).

Migration Migration of actors is triggered when one these three events occur:

– A migration message is processed (sent by another actor in the computation).

– A decision agent determines that the actor should be migrated.

– A soft failure happens in a theater. This causes the corresponding actors to migrate to any available theaters.

Split/Merge Split and merge affect the granularity of the actor system. A static number of actors will only scale so far. Splitting actors into finer granularities will allow for the system to take a computation and make it scalable to the amount of resources as more nodes join the computation. Merging allows for the system to combine actors together as resources leave the computation and the overhead of additional actors is not needed.

Replication Hard failures of nodes can cause parts of a computation to be lost, preventing its completion. Replication allows for multiple instances of the same actor to be created, to allow for fault tolerance when hard failures occur, preserving the computation.

Message Priority Message priorities are a necessity in that when an actor is in a poor location, it's mailbox can become flooded with messages. Therefore, migration messages from the middleware should have the highest priority. This ensures that these migration messages will be the next message processed by an actor, vastly improving the speed at which actors are propagated throughout the network.

3 Architecture of the World-Wide Computer

The implementation of actor mobility and autonomous actor reconfiguration consists of three software components. 1) SALSA provides a programming language for applications development, 2) WWC provides for other services including naming and message sending, along with a distributed execution environment, and 3) IO provides a middle layer set of services for autonomous actor reconfiguration, such as resource profiling for actors and theaters, and autonomous load balancing of actors across theaters (see Figure 2).

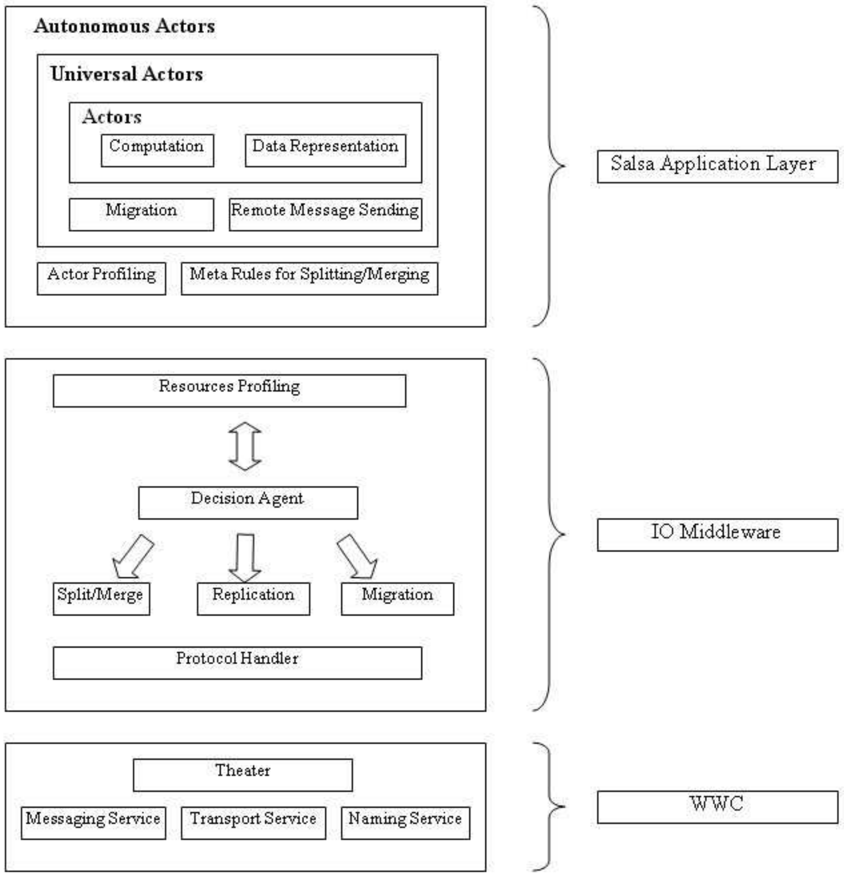


Fig. 2. Architecture of the WorldWide Computer

3.1 SALSA Programming Language

SALSA [2] programmers define behaviors much in the same way that Java programmers define a class that implements the Runnable class to control threads. Each behavior contains message handlers for manipulating the state of an actor and its thread of execution. The language support for coordination includes a variety of continuation primitives. The programming language component, SALSA, provides primitives for message passing. Messages are passed asynchronously between actors, each of which is the instantiation of an actor behavior. The language also includes migration primitives for actor mobility.

SALSA programs may be executed in heterogeneous distributed environments. During the compilation process SALSA code is compiled into Java. This precompilation allows SALSA programs to employ components of the Java class library. It also provides for a homogenous environment for distributed execution.

3.2 Theaters and Run-Time Components

The World Wide Computer architecture provides a framework that enables distributed computing over the internet. Components of the framework communicate using Universal Actor Naming Protocol (UANP) and Remote Message Sending Protocol (RMSP). RMSP enables message sending and actor migration. UANP provides for communication with naming authorities that map actor locations to human readable names.

Theaters are nodes of the WWC that provide execution environments to actor. Each theater consists of an RMSP server, a mapping between relative actor locations and executing actor references, and a runtime environment. The theater might also contain stationary environmental actors that provide access to stationary resources such as standard output.

The WWC provides a naming service which enables messages to be sent to an actor as its location changes from one theater to another. The universal actor naming protocol consists of a small number of message types and is very extensible.

3.3 Middleware for Autonomous Reconfiguration (IO)

IO builds upon the theaters of the WWC by allowing theaters to monitor system resources. IO also allows for theaters to use plug-in middleware components. The middleware components consist of:

- A Protocol Handler to allow for inter-theater communication, which provides methods for gathering information about the network topology.
- A Resource Profiling Component that gathers information about the actors' communication topology, the network topology, and the available resources in the theaters.
- A Decision Agent which decides placement of actors in the worldwide computer; given information from the Resource Profiling component. The decision agent also provides methods to determine when actors should split, merge and replicate.

These components make IO a highly reconfigurable system which is not restricted to any certain method of load balancing or fault tolerance. Therefore IO provides a testbed for comparing multiple methodologies for load balancing and fault tolerance as well as enabling hybrid load balancing schemes.

4 Strategies for Autonomous Load Balancing

In this section, we describe various methodologies for load balancing that vary by the amount of profiling done and the complexity of the decision agents. The simplest decision agents take into account the load of the individual theaters and autonomous actors, while the more complex agents consider additional factors such the network and actor topologies. All the network protocols are peer-to-peer in nature to allow for maximum scalability.

In all cases, a theater joins the autonomous network by registering with a peer server and receiving addresses of other peers in the network from it. Peer servers are not centralized, as many may serve addresses of peers for a single network.

4.1 Load Balancing Concepts

Before describing the strategies for autonomous load balancing, the following concepts are used to describe the attributes of an actor system over a heterogenous network.

Actor Satisfaction Actor satisfaction is a measure of an actor's ability to process and send messages. If an actor is not satisfied, it cannot process messages as quickly as it is receiving them. This includes the cost of message sending, because processing a message also involves sending messages. When an actor is unable to handle the load of messages it is receiving, the size of its message queue begins to increase. An actor with an increasing message queue is unsatisfied.

Theater Load Every theater hosts a group of active actors. A theater is considered lightly loaded if all its actors are satisfied, whereas a theater is considered heavily loaded if at least one of its actors is not satisfied.

4.2 P2P Random Stealing (RS)

The simplest strategy is based on random work stealing, a simple but effective algorithm described by [9]. We modified this algorithm to work in a peer-to-peer network by randomly propagating a random steal packet over the network. A lightly loaded theater chooses a neighbor at random and sends it a steal packet. This continues from theater to theater until a candidate for migration is chosen or the packet's time to live has been reached. When either occurs a notification is sent back to the originating theater. This prevents a theater form performing multiple steals simultaneously. One benefit of random steal propagation is that it avoids costly broadcasts to the network, reducing the impact of the middleware on the application. In RS, a peer theater finds its first unsatisfied actor (if one exists) and selects that as its candidate for migration. Also, since only lightly loaded theaters send steal packets, with high loads the overhead for RS becomes almost non-existant.

4.3 Actor Topology Based Random Stealing (ARS)

Actor topology based random stealing builds on the previous strategy by using additional profiling information. Actors monitor the number of messages they send to remote theaters, allowing this strategy to find a good actor topology for the network. This approach addresses the issue of the difference in expense between local and remote message sending. ARS uses a decision function to locate actors which communicate highly between each other close together.

The decision function estimates the increase in throughput an actor would receive if it migrates to a specific foreign theater. Random steal packets now also contain the available processing power of their origin theater. Let $\mathit{\Gamma}(l, f, a)$ denote the total gain that results from migrating actor a from the local theater l to the foreign theater f. The total gain is determined from the gain of communication, $\mathit{\Gamma}_c$ and the gain from processing, $\mathit{\Gamma}_p$. Migration happens only when the gain is positive ($\mathit{\Gamma}(l, f, a) > 0$). The following equations illustrate how the decision function is calculated (Refer to table 1 for notation).

$$\mathit{\Gamma}(l, f, a) = \mathit{\Gamma}_p + \mathit{\Gamma}_c - 1 \tag{1}$$

Where:

$$\mathit{\Gamma}_p = \frac{\mathcal{P}_a(f)}{\mathcal{P}_u(l)} \tag{2}$$

and

$$\mathit{\Gamma}_c = \frac{\mathcal{N}_c(f) - \mathcal{N}_c(l)}{\mathcal{N}_p} \tag{3}$$

This decision function was chosen because while it is not very precise, it does provide very reasonable results with a minimal amount of overhead. It places a strong emphasis on inter-actor communication and tries to colocate tightly coupled actors (actors which frequently communicate). It also gives appropriate results in that a large processing gain can outweigh actor coupling, and vice versa.

Notation	Explanation
$\mathcal{N}_c(t)$	The number of messages communicated between an actor and theater t
$\mathcal{N}_p$	The number of messages processed by the actor
$\Gamma(l, f, a)$	The gain obtained by migrating actor a from theater l to theater f
Γ_c	The gain obtained from actor's comminication
Γ_p	The gain obtained from message processing
$\mathcal{P}_a(t)$	The processing power available in theater t
$\mathcal{P}_u(t)$	The processing power used in theater t

Table 1. Notation Used in the Decision Function Equations.

4.4 Network Topology Based Random Stealing (NRS)

In addition to resource availability, NRS takes into consideration the topology of the network. In the IO network a peer might belong to local, regional, national, or international clusters [10]. In these cases, while bandwidth may be high, latency will play a large factor in the throughput of messages between theaters. NRS locates tightly coupled actors close together in the IO network, but allows loosely coupled actors to migrate more freely, as they do not need this restriction.

NRS classifies its neighbors into four groups: local, regional, national and international. These groups are classified into locales by the following ping times [10]:

- Local: 10 ms or less
- Regional: 11 ms to 100 ms
- National: 101 ms to 250 ms
- International: 251 ms and higher

The algorithm then proceeds similar to cluster-aware random stealing described by [11]. Random steal packets specify which locale they are to travel. A theater first selects a local peer randomly and sends a local random steal packet. A theater will only propagate a steal packet to its specified locale. If a local random steal packet fails (the theater receives a terminated packet without an actor), the theater will then attempt a regional random steal, and so on.

Using this method to propagate random steal packets through the network keeps groups of coupled actors close together in the network. NRS uses the previously mentioned methods for determining the best candidate actor when a random steal packet reaches a theater, thus NRS comes in two versions: RS and ARS.

4.5 Fault Tolerance Strategies

Currently, all the decision agents use a round-robin strategy to disperse their actors to all their available neighbors in the case of a soft termination of the theater. In this case, the goal is not to optimize actor placement, but rather to complete the termination process as soon as possible before the hard theater failure.

5 Preliminary Results

We ran a series of tests on our IO system using a manual round robin placement of actors (RR), peer-to-peer random stealing (RS) and the actor topology based random stealing (ARS) strategies.

We ran four simulations each pertaining to a level of inter-actor communication. The unconnected actor graph had actors simply process messages over and over, with no inter-actor communication. The sparse actor graph linked actors randomly, providing a moderate amount of inter-actor communication. The tree simulation linked actors in a tree structure, for a higher amount of inter-actor communication. Lastly, the hypercube provided a very high amount of inter-actor communication. (see Figures 3, 4, 5 and 6). We compared throughput of RS and ARS to manual load balancing to measure the overhead that the IO middleware incurred on the computation. All actors were loaded in a round robin fashion across the eight theaters, then were allowed to compute until their throughput leveled off. Throughput is the number of messages processed by all actors in a given amount of time – the higher the throughput, the faster a computation is running.

Figure 3 shows that both ARS and RS imposed a minimal amount of overhead for the simulation, as a round robin placement of actors is the optimal load balancing solution for an unconnected graph of actors in a homogeneous network, and the round robin placement imposed no middleware overhead. ARS and RS performed comparatively to RR in this test. On the more communication-bound simulations (see Figures 5 and 6), ARS outperformed both the manual load balancing and RS. On a sparsely connected graph, ARS performed superbly, bringing throughput to nearly the level of an unconnected graph. In all simulations involving inter-actor communication, ARS highly outperformed RR and RS, showing that the co-location of actors significantly improves message throughput. RS was shown to be too unstable in all these simulations and did not outperform either RR or ARS. Our conjecture is that because the Java thread scheduling mechanism is not fair, actors are found to be unsatisfied when they are actually not, leading to the unstable migration behavior of actors when IO uses RS.

To show how IO can handle a dynamically changing network, the same simulations were ran on a changing network of peer theaters. The simulations were loaded entirely onto one peer theater, then every 30 seconds an additional peer theater was added to the computation. After eight peer theaters had joined the computation, IO was allowed to run for two minutes to balance the load, after which a peer theater was removed every 30 seconds, until the computation was entirely running on the last peer theater added to the computation.

With the unconnected graph join/leave simulation (see Figure 7), both RS and ARS performed well in distributing the load across the peer theaters (see Figures 8 and 9), and increased the throughput by a factor of about six when all eight theaters had joined the simulation. The addition and removal of peer theaters shows that IO can rebalance load with removal and addition of nodes without much overhead.

In the tree join/leave simulation (see Figure 10) ARS performed well, increasing throughput by a factor of about 3.5 when all eight theaters were joined in the simulation. RS however performed worse than simply loading the simulation entirely onto one theater. The graphs of actor placement (see Figures 11 and 12) show that while both ARS and RS managed to distribute the actors evenly across the network of theaters, ARS co-located actors more appropriately accorting do their connectivity, significantly improving overall throughput.

These preliminary results show that the IO system with ARS performs well in most situations for load balancing of an actor system. While the more traditional strategy of random stealing does not fare so well in an autonomous system of actors, a more intelligent strategy can exploit the properties of the actor model to provide autonomic solutions for load balancing across a dynamic network. The results also show that IO can handle the addition and removal of nodes from a computation without any central coordination, a necessity for large dynamic heterogeneous networks.

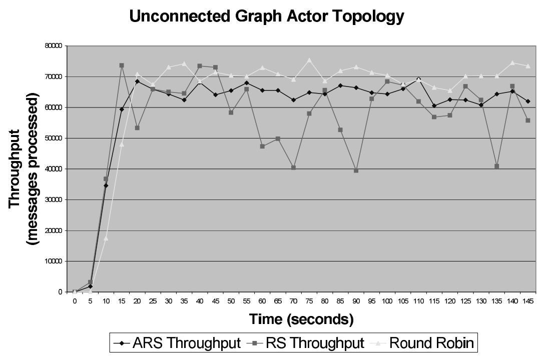


Fig. 3. Unconnected Graph Simulation

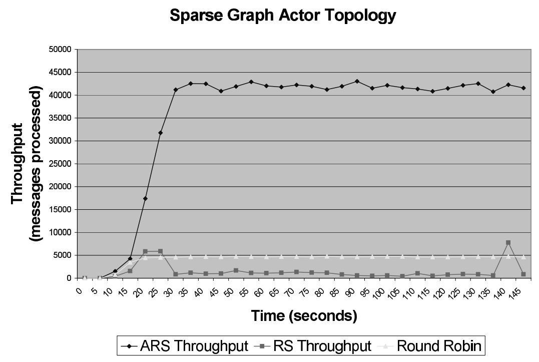


Fig. 4. Sparse Graph Simulation

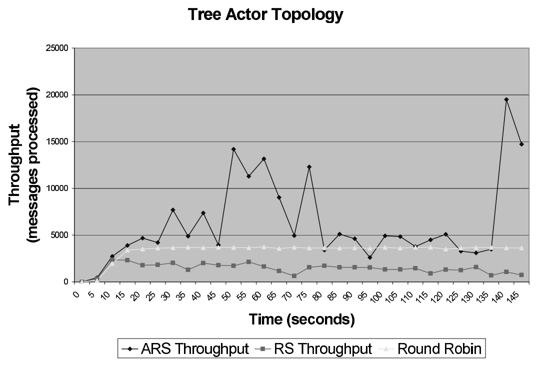


Fig. 5. Tree Simulation

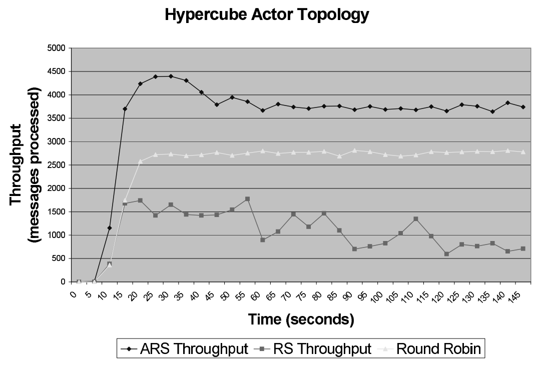


Fig. 6. Hypercube Simulation

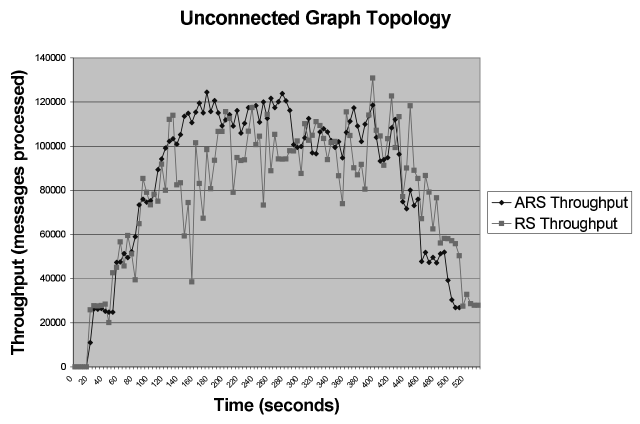


Fig. 7. Unconnected Graph on a Dynamic Network.

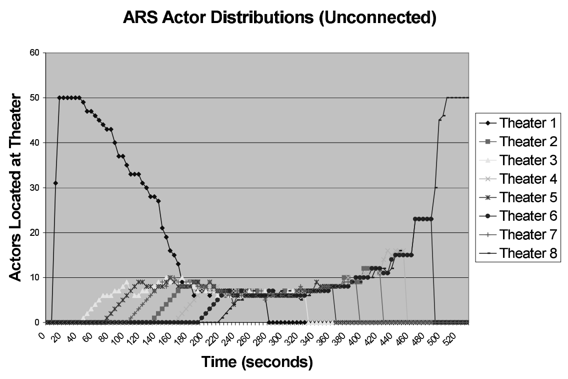


Fig. 8. Actor Distribution for ARS

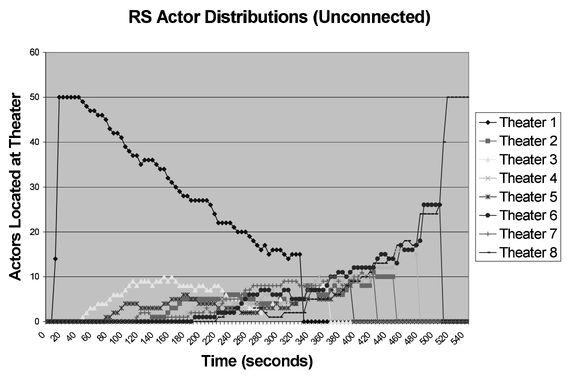


Fig. 9. Actor Distribution for RS

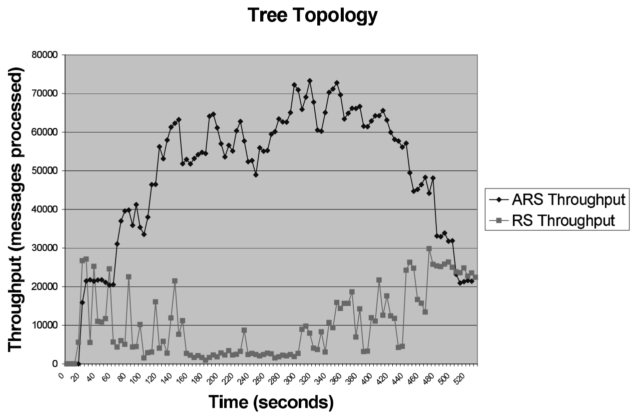


Fig. 10. Tree Topology on a Dynamic Network.

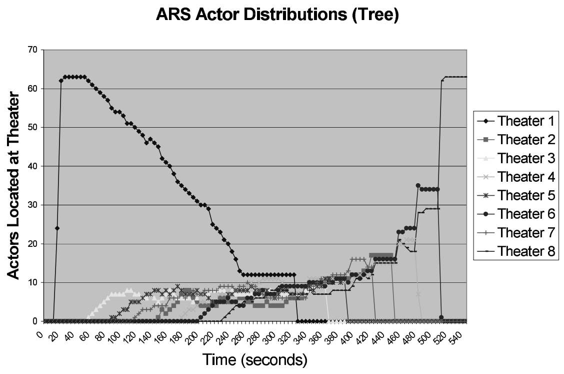


Fig. 11. Actor Distribution for ARS

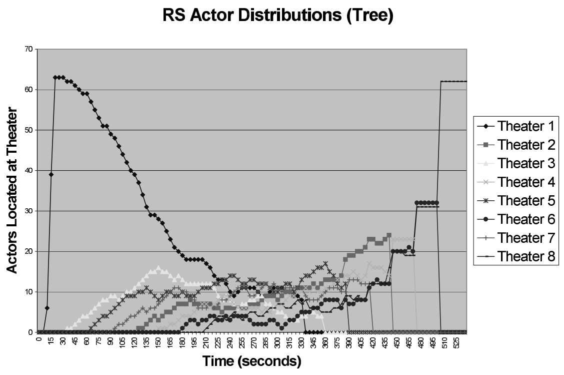


Fig. 12. Actor Distribution for RS

6 Related Work

A significant amount of research has been done on load balancing at various system levels. Network-level load balancing tries to optimize the utilization of existing network resources by controlling traffic flow and minimizing the number of over-utilized links and under-utilized links [12], while operating system-level load balancing aims at balancing the load between shared distributed resources such as processors, disks, or memory [13]. Middleware-based load balancing provides the most flexibility in terms of balancing the load to different types of applications [14]. It is not constrained to the OS or network level but it spans different system levels. Different load balancing strategies exist that range from static to dynamic, centralized to distributed, and sender-initiated to receiver-initiated strategies. For a more detailed classification of load balancing strategies, we refer the reader to [15].

This work focuses on middleware-level based load balancing strategies across large scale, highly dynamic peer-to-peer networks. Several load balancing strategies have been studied for structured and unstructured P2P systems. Some of them distribute objects across structured P2P systems [16–18]. They are all based on the concept of distributed hash tables. However they assume that all objects are homogenous and have the same size. Rao et al. [19] have accounted for heterogeneity by using the concept of virtual servers that move from heavy nodes to light nodes which is similar in concept to migration of our actors. However they assume that the load on virtual servers is stable. They also assume that there is only one bottleneck resource that needs to be optimized at a time.

Triantafillou et al. [20] have suggested load balancing algorithms to distribute contents over unstructured P2P systems. They aggregate global meta-data over a two-level hierarchy and they use it to re-assign objects.

Our work does not assume any specific P2P structure. Our load balancing decision functions are not restricted to optimizing a specific bottleneck resource. Rather, our decision agents use the nature of the actor model to determine the placement of actors in the system. Moreover, our middleware environment IO is not restricted to one load balancing strategy. It has been designed and implemented with the intention of plugging in different load balancing strategies depending on the nature of the running applications. This allows us to create concepts and decision functions based on this actor model, where placement of actors is not restricted to specific resources. We have used this middleware as a testbed to evaluate different strategies with several application communication topologies simulating diverse applications.

The issue of adaptive middleware in distributed systems have been studied by several researchers. Gul Agha et al. have introduced meta-actors to implement different interaction services such as fault tolerance, security, and synchronization [7]. Fabio Kon et al. have presented a model of reflective middelware that allows dynamic inspection and modification of the execution semantics of running applications as a response to changing resources in a distributed environment in order to improve performace [21]. Research has also been done at the level of middleware security. Venkatasubramanian discussed the safe composibility of reflective middleware services to ensure the trustworthiness of systems [22]. As future work, we are planning to adopt similar reflective strategies that allow our IO middleware to adapt more efficiently to the dynamic nature of large scale networks. The middleware should be able to integrate easily more decision functions, to choose on the fly what decision function and what profiling level to use depending on the nature of the computation, the network topology, or the actor topology.

7 Conclusion and Future Work

Our preliminary version of IO has shown that more intelligent load balancing schemes are needed to improve the perfomance of a computation across a distributed actor environment. The implemented decision function has focused mainly on profiling the CPU processing powers across theaters. Different applications have different resource requirements, therefore the amount of profiling done is highly dependent on the nature of computation that is being performed by the actors. The IO middleware should be intelligent enough to select dynamically what level of profiling should be done by the decision function. As future work, we are planning to profile more resources, such as bandwidth, memory, and storage. The developement of the World-Wide Computer is still an ongoing process. We are planning to devise and test different startegies for the split/merge and replication components. Our long term goal is to define, develop, and deploy a platform for worldwide computing that enables resource-intensive applications to locate and allocate resources and adapt to highly dynamic environments regardless of their location or platforms.

Acknowledgements

We would like to acknowledge the worldwide computing research group at Rensselaer Polytechnic Institute for many discussions about this work, the SALSA developement group for their contribution in the design and implementation of SALSA, and the IO group for their fruitful discussions and continous feedback regarding the IO project. In particular, we would like to thank Abe Stephens for careful readings of previous drafts of this paper. Any errors/omissions remain our own.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
2. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices **36** (2001) 20–34
3. Kim, W., Agha, G.: Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In: Proceedings of Supercomputing'95. (1995)
4. Agha, G., Jamali, N.: Concurrent programming for distributed artificial intelligence. In Weiss, G., ed.: Multiagent Systems: A Modern Approach to DAI. MIT Press (1999)
5. Tomlinson, C., Cannata, P., Meredith, G., Woelk, D.: The extensible services switch in Carnot. IEEE Parallel and Distributed Technology **1** (1993) 16–20
6. Ren, S., Agha, G.A., Saito, M.: A modular approach for programming distributed real-time systems. Journal of Parallel and Distributed Computing **36** (1996) 4–12
7. Agha, G., Frølund, S., Panwar, R., Sturman, D.: A linguistic framework for dynamic composition of dependability protocols. In: Dependable Computing for Critical Applications III, International Federation of Information Processing Societies (IFIP), Elsevier Science Publisher (1993) 345–363

8. Ferber, J., Briot, J.: Design of a concurrent language for distributed artificial intelligence. In: Proceedings of the International Conference on Fifth Generation Computer Systems. Volume 2., Institute for New Generation Computer Technology (1988) 755–762
9. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. In: Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94), Santa Fe, New Mexico (1994) 356–368
10. Kwan, T.T., Reed, D.A.: Performance of an infrastructure for worldwide parallel computing. In: 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, San Juan, Puerto Rico (1999)
11. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. ACM **36** (2001) 34–43
12. Saito, H., Miyao, Y., Yoshida, M.: Traffic engineering using multiple multipoint-to-point LSPs. In: INFOCOM (2). (2000) 894–901
13. Krebs, W.G.: (Queue load-balancing/distributed batch processing and local rsh replacement system)
14. Othman, O., Schmidt, D.C.: Issues in the Design of Adaptive Middleware Load Balancing. Proceedings of the 2001 ACM SIGPLAN workshop on Optimization of middleware and distributed systems (2001)
15. Krahl, R., Nolte, J., Bttner, L.: (A load balancing approach for the peace operating system)
16. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings of ACM SIGCOMM 2001. (2001)
17. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM '01 Conference, San Diego, California (2001)
18. Hildrum, K., Kubiatowicz, J.D., Rao, S., Zhao, B.Y.: Distributed object location in a dynamic network. In: Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures. (2002) 41–52
19. Karthik, A.R.: Load balancing in structured p2p systems (2003)
20. Triantafillou, P., Xiruhaki, C., Koubarakis, M., Ntarmos, N.: Towards high performance peer-to-peer content and resource sharing systems (2003)
21. Kon F., Costa F., B.G., H., C.R.: The case of reflective middleware. Commun. ACM **45** (2002) 33–38

22. Venkatasubramanian, N.: Safe composibility of middleware services. Commun. ACM **45** (2002) 49–52