修　士　論　文

A High-performance Deadlock-free Overlay for Wide-area Parallel and Distributed Programming

広域並列分散プログラミングのための高性能デッドロックフリーオーバレイ

指導教員　田浦健次朗 准教授

東京大学 情報理工学系研究科
電子情報学専攻

氏 名　076426 弘中健

提 出 日　平成21年2月4日

Abstract

Parallel and distributed computing in Wide-area environments is complicated by connectivity issues like NATs/firewalls, and dynamic changes in the available resources. Therefore, programming in these environments requires substantial support from programming frameworks. Such frameworks can help resolve low-level concerns while providing abstractions to implement applications more easily. Meanwhile overlay networks, which create their own network on top of the internet using UDP/TCP links, have become popular communication mediums for frameworks in these environments as they transparently enable communication even in wide-area networks. Thus, high-performance overlays are crucial in this context. This paper presents one such programming framework and a high-performance overlay whose goal is to serve as its communication layer.

The first proposal designs and implements an overlay-based programming framework for large-scale wide-area computing by minimally extending distributed object-oriented models for maximum *generality* and *flexibility*. Aside from introducing constructs to facilitate parallel computation and to handle runtime process joins/leaves, the framework automatically creates and manages an overlay to enable communication across WANs with minimal user intervention. In the framework, parallelism is expressed via asynchronous method invocations to distributed objects for a natural transition from sequential programs. To make parallelism manageable, the framework introduces an implicit serialization semantics on objects to relieve programmers from explicit synchronization, while avoiding deadlock problems. In the implementation, participating nodes automatically construct an overlay, and relieve the users from manual configuration and managing the overlay as it dynamically changes during the computation. This framework, gluepy is implemented as a library for Python to allow rapid development of complex workflows and to maximally leverage the richness of its libraries. For evaluation, this work shows, on over 900 cores across 9 clusters with complex networks (involving NATs and firewalls), how applications with dynamic node joins/failures can be expressed simply and executed easily.

The second proposal presents the design and implementation of a high-performance overlay network that attains the throughput performance of the underlying wide-area network. The addressed core issue is a flow control problem where intermediate overlay nodes have limited buffer memory, while the forwarding must yield full network throughput exceeding Gbps. Implementing a naive flow control, however can deadlock the overlay. The proposed overlay presents a deadlock-free overlay that couples TCP connections and fixed intermediate buffer memory. The method fully takes advantage of TCP's flow control and implements a simple flow control scheme by creating dependencies among connections in an overlay path. Meanwhile, the proposal adapts a deadlock-free routing algorithm for heterogeneous wide-area networks, so deadlocks among communication can be avoided without sacrificing performance. The proposal also incorporates overlay construction and routing optimizations that account for underlying network latency and bandwidth information. This work demonstrates, via simulation on 13 clusters (515 nodes) and by evaluation on 7 clusters (170 nodes), that the proposed deadlock-free routing poses negligible overhead in comparison to deadlock-unaware routing, and comparably with direct communication. It further demonstrates that for certain collective communications, the proposed overlay even out-performs direct communication by mitigating or completely avoiding network contention. This is shown on systems ranging from a single-switch cluster with 36 nodes to a Grid environment with 4 clusters and 291 nodes.

Acknowledgement

First of all, I would like to express my sincere appreciation to my supervisors Professor Takashi Chikayama, and Professor Kenjiro Taura for their continuous support and advice for the past three years. Professor Chikayama gave advice that was on the spot, while always leaving room for wit. I especially would like to thank Professor Taura for his lessons on and off the field. The divine lessons in Python shall never be forgotten. I would also like to express my deepest respect for this man. I haven't seen anyone in my life, who has been able to turn down a banquet dinner in Lyon, the city of gourmet, for a value meal at a McDonald's. Some would say, disinterest for food. I say, commitment to his work. Finally, I am deeply grateful to my supervisors for leading me into this extraordinarily interesting research field. While my immediate path does not take me towards research, it has undoubtedly shaped the person who I am today, and the career vision I intend to follow in years to come and many more.

I thank all my colleagues in my research group for making this journey as exciting and enjoyable as it has been. It has truly been a great honor to be able to work amongst such talented and devoted group of people. I would like to express my gratitude to the 2 research assistants, Yoshikazu Kamoshida, and Daisaku Yokoyama, for being so patient and always willing to give assistance. I also thank the graduate, Kei Takahashi for his optimism and creating such a comforting atmosphere to this research lab, in addition to his insightful research advice. I thank Masaaki Yasui, for our deeply meaningless discussions. I have been able to maintain my sanity because of you. I am also grateful to Yuto Hayamizu for the constant tech trend updates, not to mention his generous servings of coffee. I am deeply indebted to Hideo Saito, not only for being such a great research mentor, but also for being such a supportive friend through this journey. It was a great pleasure to work in collaboration with you in research, in server administration, and in other probably not so important stuff.

I also like to thank InTrigger, the research Grid platform I, as a team, have helped develop and grow. You in turn, have helped grow in many ways as well. In some aspects, I have spent the most time with you, with sleepless nights administrating and debugging, and with again, sleepless nights debugging my research code. You have served as a great mentor and sandbox where I honed my abilities as a system adminstrator, and as a Computer scientist.

I also like to thank Thorsten Strufe and Michael Rossberg for making my training in Ilmenau as multi-dimensional and rewarding as it has been. I shall always follow the ways of the “Hornerwhisky”.

Finally, but not least, I would like to thank all my friends and family who have supported me in all aspect in surviving the past three years. In particular, I would like to thank Kirsty Lebsanft and my parents for their unconditional support. I surely would not have made it without your support, mentally and physically.

February 10, 2009

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Computation in wide-area environments

The performance increase of commodity computers and wide-area networks has widened the field of application, as well as increased the scale for parallel and distributed computation. Notable platforms for such computation include TeraGrid [13] in the United States, the Grid5000 in France [5], the DAS-3 [3] in the Netherlands, and InTrigger [7] in Japan. These environments are Grids, and are composed of clusters. Intra-cluster compute nodes are connected in local area networks (LANs), and clusters are connected over a wide-area network (WAN). Until recently, wide-area computing has been reserved to applications that require large amounts of computation but coarse-grained and minimal communication like volunteer computing [4, 20] and embarrassingly parallel computation. However, with the emergence of these platforms and high-speed wide-area network backbones [10, 11], parallel and distributed computing has also become increasingly more prevalent [39, 55]. Such applications, in addition to requiring intensive computation, require fine-grained and closely coupled communication among participating computers [25, 42, 65].

1.1.2 Issues with computation in large-scale wide-area environments

Though these Grid platforms can potentially deliver unprecedented scales of computation and application, they introduce problems particular to wide-area networks.

connectivity: In many cases, clusters belong to different administrative domains. Thus, security and management policies can differ greatly across clusters used. For network environments, this manifests itself in the form of firewalls and private IPs/NATs (Network Address Translations). These barriers hinder 2 arbitrary compute nodes from directly communicating to each other.

This may happen symmetrically, where the node pair cannot communicate directly at all, or asymetrically, where communication may only be initiated from one side.

heterogeneity: While compute nodes in LANs are interconnected via low-latency, wide-bandwidth networks such as Gigabit-Ethernet, Myrinet [27], and Infiniband [6], WAN communication in contrast has high-latency and tends to have narrow-bandwidth. Thus communication in WANs must endure large discrepancies in latency (in the order of [us] to [ms]), and in bandwidth (in the order of 10[Mbps] to 10[Gbps]). Heterogeneity across different clusters must also be accounted for. Each cluster is subjected to different policies and resource management software. Because of this, users are forced to adapt to a variety of tools and rules to utilize resources on multi-cluster environments.

scalability: In clusters of clusters computation, the scale of computational resource can reach hundreds and even thousands of compute nodes. This is challenging especially for communication due to scalability limits of stateful routers, file descriptor and/or memory liminations of individual computing processes, and network bottlenecks.

dynamicity: Because the scale of computation and network resources is large, and each portions of the environment are affected by varying policies and problems, resources in a large-scale wide-area environment are dynamic. Network and compute node failures are a given. Wide-area networks are unstable by nature, and even entire clusters may go down for maintenance.

1.2 Motivation

1.2.1 Motivation Overview

Problems introduced above make parallel and distributed computing in large-scale wide-area environments very complex from the programming standpoint. Therefore, programming in these environments requires substantial support from programming frameworks. These frameworks must resolve such low-level concerns transparently from the programmer. They must also provide good abstractions and programming models to express parallelism easily, while not being being too rigid to limit their application.

These programming frameworks in turn, require a high-performance communication medium so they can maximize the performance and scalability of the applications above. Meanwhile, overlay networks are gaining popularity in wide-area environments, because they can address WAN connectivity issues as well as scalability with respect to network resources. They are particularly suitable for implementing the above frameworks as they can easily provide connectivity and communication transparency to the user. Thus, the performance of these overlays in these environments are of particular concern.

The remainder of this section discusses these two motivations in more depth and clarify the necessary qualities. They will serve as an outline in understanding the contributions of this work.

1.2.2 Parallel and distributed programming frameworks for wide-area environments

The foremost task of programming frameworks for these environments is to provide comprehensive programming models and abstractions to express computation and parallelism so the above complexities are made manageable in a simple and uniform framework. Thus, a programming framework for large-scale wide-area environments must address (among others) the following characteristics:

- Allow processes to simply and seamlessly communicate across sites despite the complexity of underlying networks
- Incorporate dynamically joining processes into the computation while providing means to handle node and network failures
- A programming model to express and handle parallelism simply and concisely

A common approach has been to design a framework that *hides* these issues, but in doing so, restricts the programmer to stick to a very rigid model. Good examples are systems and frameworks for embarrassingly parallel applications [8, 24, 62] and their extensions to express dependencies among tasks [2, 12]. They normally require no programming effort for coordinating parallel execution on a distributed environment. A slightly more general approach has been to design programming frameworks specific to certain application domains [31] and coordination models [33]. These frameworks hide communication and dynamic resource concerns, and are perfect when the problem at hand naturally fits their model. However, all of the above approaches usually provide no or limited means for individual tasks to communicate with each other. Implementing the coordination among subtasks must frequently resort to using "out-of-band" means (e.g., file transfer, ad-hoc CGI, etc.), making the code awkward and error-prone. The users' programming flexibility as a result, is sacrificed and it becomes hard to design a wide range of parallel and distributed applications.

1.2.2.1 This paper's approach: Framework that retains programming flexibility

Another approach, which is pursued in this paper, is to leverage a general-purpose programming language and conventional parallel/distributed programming concepts, and minimally extend them for issues in large-scale wide-area environments. Extending on such languages retains the users' programming flexibility. Object-oriented languages are particularly suitable for this purpose, as they have a general and an accepted model of communication (i.e., remote method invocation or RMI for short).

Yet existing parallel RMI-enabled frameworks [37, 64] do not sufficiently address the aforementioned issues (connectivity, scalability, and dynamic processes). Furthermore, managing parallelism and asynchroneity with distributed objects is still not trivial, and race-conditions and deadlocks are commonplace. Thus, this work designs and implements a new programming framework with addressing these issues as its primary objective.

1.2.3 Application level overlays for wide-area parallel and distributed computation

The internet realizes point-point communication between nodes by IP routing. Application level overlays in contrast, shown in Figure 1.1 use such point-point communication as links for another

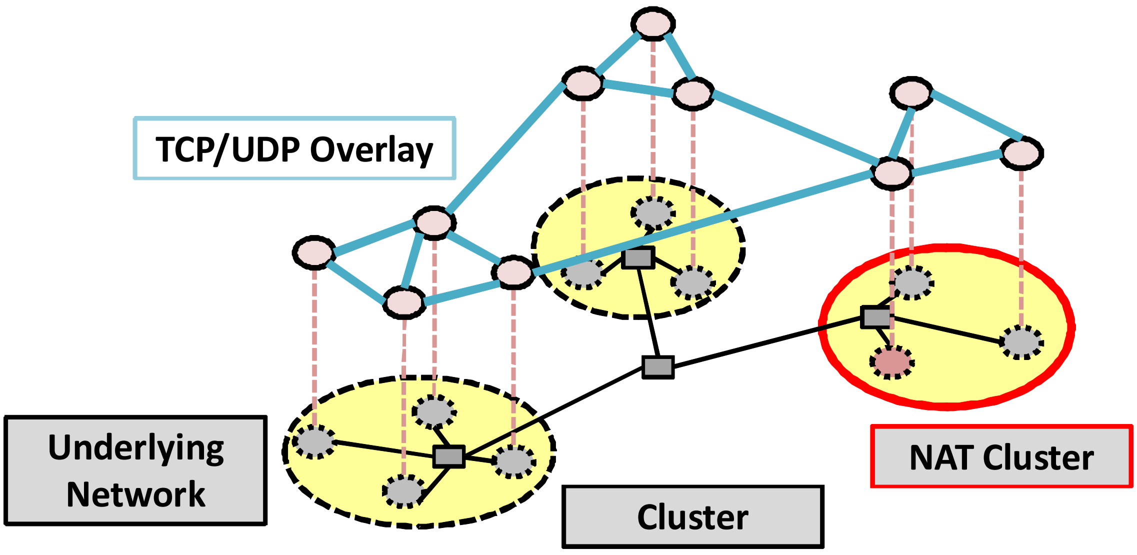


Figure 1.1: Application-level overlay

network built on top of the internet. Specificly, they create another network graph where each link is a TCP or UDP link, which in turn is realized by the IP layer. Overlays perform their own application level routing independently of that of the underlying internet. In wide-area environments, application level overlays have been adopted to overcome communication NATs/firewalls restrictions [47]. 2 nodes that cannot direct communicate to each other can find a common node, to which it is connected to on the overlay, and route communication via that common node. They have also been utilized to address scalability limits of stateful routers, and file descriptor and/or memory limitations of processes by purposefully creating a sparse connection graph for all point-point communication rather than a complete graph had direct communication been required [55].

1.2.3.1 Overlays for wide-area parallel and distributed programming frameworks

Application level overlays are particularly important in the context of wide-area parallel and distributed computing. Overlay networks enable computing nodes distributed across multiple clusters to communicate with each other. Overlays are also important because in wide-area distributed computations, the number of computing nodes are large and thus the communication scalability becomes a critical concern. Moreover, overlays are favorable because distributed programming frameworks can implement them to enable transparent communication over WANs and shelter the application programmer from the low-level communication and connectivity concerns. Thus, overlays are becoming indispensable communication mediums for large-scale parallel and distributed computing on wide-area environments.

1.2.3.2 Performance incentives to adopt overlays

Until recently, overlays have largely been viewed as necessary workarounds used *only* when nodes are otherwise unable to communicate because they introduce extra hops and overhead. But in the context of high performance parallel and distributed computing, overlays will become necessary not only to enable communication among otherwise blocked node pairs, but also to deliver *better* performance than that by making as many direct connections as possible. Improved communication performance by overlays has been demonstrated in other contexts other than parallel and distributed computing [18, 19, 45, 51], but their primary motive was to avoid bad paths by the underlying IP routing, and ensure that the communication will take IP layer paths that deliver smaller latencies using a series of UDP/TCP links. As it will be shown in this work, overlays can achieve performance improvements by purposely holding back concurrent communication, and mitigate or completely avoid traffic contentions. This is especially important for parallel and distributed computing as overlays can yield significant performance improvements for commonly used collective communication operations. This is not reserved to wide-area/heterogeneous environments, but even for networks with high speed interconnects like a LAN. Thus, the design and implementation of high-performance overlays will become a very important topic in the context of high-performance parallel and distributed computing.

1.2.4 High performance overlays for wide-area computing

1.2.4.1 Problem of achieving high throughput in overlays

In order for overlays to replace raw sockets as the communication medium for high-performance parallel and distributed computing, it must provide (among others) high throughput performance in LAN/WAN-mixed environments. Yet one issue that must be addressed in any overlay, but has largely been overlooked, is flow control and memory management for router nodes in the overlay. Specifically, it refers to an algorithm for the router nodes that feed and forward incoming traffic to transfer data from the source to the destination node. The standard "select-loop", which receives all readable data and buffers them in memory until the outgoing connection becomes writable, is prone to buffer memory overflow. This issue is becoming increasingly critical in high speed networks, where 10[Gbps] links can fill as much as 1GB memory in a second, and in local/wide-area mixed computation where discrepancies among input/output link bandwidths are everywhere.

There have been some efforts to implement flow control schemes for overlays, but they are complex and do not yield high performance. All previous schemes in effect, first build an *unreliable* overlay and implement TCP-like end-to-end flow control on top [16, 23, 38]. Such schemes may be natural if the overlay uses unreliable underlying transports (such as UDP). This choice is in turn natural if the overlay does not provide reliability to its user; deadlock can be easily avoided in such frameworks (at least superficially) by dropping packets at intermediate nodes when buffer memory is full. To provide congestion avoidance or reliability, they introduce additional overhead by requiring periodic communication among nodes to secure the correct behavior of the protocol. In addition, they require non-trivial parameter tuning to adjust to the performance and conditions of underlying networks, essentially reimplementing what TCP already provides.

1.2.4.2 Deadlocks problems with simple flow control schemes

A very simple flow control solution is to use *back-pressure*, which is to block the sender nodes when the intermediate buffer becomes full. Such a solution, if naively implemented, is prone to *communication deadlock* within the overlay. Such deadlocks are cases where a set of data transfers on the overlay can no longer make progress. This is possible when there is a cyclic dependency among the set of data transfers. For example, a data transfer A blocks because buffers for subsequent nodes in the overlay path are full. This can be caused by a different data transfer B, whose data is occupying those buffers. In such a case, transfer A cannot proceed until transfer B makes progress and makes room at buffers that A needs, and thus transfer A is dependent on transfer B: denoted here as $A \rightarrow B$. In a general overlay, where multiple transfers co-exist, there are multiple dependencies among transfers. It is possible that such dependencies result in a cycle, where for example 4 transfers A, B, C, D have dependency as follows: $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. None of the 4 transfers can no longer make progress, they have created a deadlock.

Despite its importance, this problem has been paid surprisingly little attention in the context of overlays. This is presumably because routing has been conceived as something that should be avoided as much as possible, and because only simple topologies (such as having a single hub node for all inter-cluster communications) , which need no sophisticated solutions to avoid deadlocks, have been employed. The situation is changing however, as we now commonly have three or more clusters, and more nodes within each cluster [3, 5, 7].

1.2.4.3 This paper's approach: Deadlock-free high performance overlay using TCP

In high performance computing, using TCP as an underlying transport, in comparison to using UDP and completely reimplementing what TCP provides onto the overlay, is a more natural choice because it gives reliability as well as high-performance point-to-point flow control [47, 55]. It has been well-tested to work well on a wide variety of systems and networks, ranging from modem lines to 10[Gbps] Myrinet and Infiniband links. Yet, naively using TCP for overlays is either memory overflow- or deadlock-prone. So this leads to the question: is it not possible to retain the advantages provided by TCP itself, while making the transition from direct communication to application overlays? For certain fixed communication patterns such as broadcast, coupling TCP connections with fixed intermediate node buffers can achieve flow control across multiple dependent TCP connections [44, 63]. The missing formula is how to prevent communication deadlocks for general unknown traffic patterns, while delivering high performance of the underlying transport to the application as much as possible.

To this end, this work present how a flow controlling overlay can be implemented using TCP and *deadlock-free routing* [28, 30, 57]. Simply deploying deadlock-free routing on an arbitrary TCP overlay, however, is not sufficient to address performance issues for parallel computation. It presents the following challenges.

Considerations for the underlay: The overlay and the routing must take the underlying network topology and bandwidth into consideration during construction.

Overhead: While deadlock-freedom is imperative, the extra overhead of using deadlock-free routing must be minimal.

1.3 Contributions of this Work

The contributions of this work come in 2 folds as presented below. The goal of this work is to present a simple and flexible programming framework for parallel and distributed computation in a large-scale dynamic wide-area environment, with a high throughput underlying communication layer that accounts for the heterogeneity of wide-area networks.

1.3.1 gluepy: A distributed programming framework for wide-area environments

This work presents *gluepy*, a Python distributed object oriented programming framework. The framework enables simple, concise, and flexible parallel and distributed programming in large-scale wide-area environments with seamless communication even in environments with NATs/firewalls and with dynamically joining/leaving processes. The core aim of this work is to provide a simple scripting library to *glue* together processes for parallel and distributed computing and to overcome network barriers and compute resource dynamics, while still giving access to the wealth of libraries and the simplicity of Python programming.

The framework comes with the following contributions:

- Objects have implicit serialization semantics. Parallelism is expressed via asynchronous procedure/method calls, using primitives commonly known as *futures* [61]. Yet the semantics eliminate the need for explicit synchronization code. The underlying execution model is still based on passive objects + threads familiar to many programmers. The proposed design thus does not suffer from the self-recursion deadlocks that some active object-based models do [14, 37].
- An object signalling mechanism that provides simple means by which asynchronous events, such as new process joins, may be handled. It is designed so as to retain the comfortable programming style of using asynchronous method invocations to manage and schedule tasks without resorting to low level event-handling loops.
- A TCP overlay network is built among participating nodes to realize scalable and seamless communication among all participating nodes. By default, the framework requires no configuration and builds a connected overlay for WANs. To incorporate resources only reachable via SSH, it can use SSH port forwarding where specified. The necessary configuration for this change is minimal, merely 1 line in the configuration file.

The experimental platform includes 9 clusters with over 900 CPU cores. Many of them perform IP filtering to various degrees, and some only have private addresses. It also includes an almost completely confined cluster that can be reached only by first logging in to its gateway via SSH, then logging in to a cluster frontend via SSH, and finally submitting jobs via a batch scheduler. The setting required for collectively using all resources in the environment was 3 lines in a configuration file. The main result of this work is a simple scripting environment that can coordinate processes spread across such complex environments. With little effort, a combination optimization problem solver was implemented and deployed on a platform of over 900 CPU cores. The core of the solver is an optimized sequential program written in C++ that won the 3rd Grid Plugtest Contest [1]. Python code in the proposed framework merely schedules the underlying C++ processes and exchanges solutions found among participating processes. The glue code is very concise and has only 250 lines, the core of which is shown in Section

1.3.2 High-performance wide-area overlay using deadlock-free routing

This work proposes a wide-area, network locality-aware overlay network, and uses a simple deadlock-free forwarding mechanism to straightforwardly attain the throughput performance of the underlying network for point-point communication, while avoiding network contention and congestion to deliver performance improvements for collective communication. The aim of this work is to provide a high-performance communication medium for large-scale wide-area parallel and distributed applications/middlewares.

Specificly, it makes the following contributions:

- show a simple and high performance implementation of deadlock-free overlay routers that works with constant memory per connection and can straightforwardly attain the performance of the underlying reliable transport (TCP in this paper).
- show the implemented scheme actually achieves the bandwidth close to the underlying TCP over a range of environments including GbE, and 10G Myrinet.
- propose an algorithm for choosing good routing paths in heterogeneous wide-area environments, as well as one for choosing overlay links in the first place, given suitable information about the underlying network [48, 58]. As a result, deadlock-prone routing and deadlock-free routing paths have very little performance differences.
- demonstrate such an overlay scheme outperforms direct connections when contentions occur at bottlenecks (even in LAN).

The evaluation was conducted both by simulation and on real wide-area environments. On 515 simulated nodes (13 clusters) from the InTrigger platform [7], it was demonstrated that the proposed deadlock-free mechanism has very little overhead in comparison to deadlock-unaware routing. The overlay was constructed on cluster environments with various network throughputs (100[Mbps], 1[Gbps], 10[Gbps]), on a 7-cluster heterogeneous wide-area environment, and demonstrated that its throughput and latency is comparable to that of direct communication. Furthermore for gather and all-to-all collective communications, the overlay outperformed direct communication on environments ranging from a single switch LAN with 36 nodes to a 4-cluster environment with 291 nodes.

1.4 Organization of this Paper

The rest of the paper is organized as follows. Chapter 2 discusses related work with respect to both programming frameworks and application overlays in large-scale wide-area environments. Chapter 3 presents the proposed programming framework. Chapter 4 presents the proposed deadlock-free routing overlay. Evaluation for both the programming framework and the overlay are given in Chapter 5. Finally, concluding remarks and future works are presented in Chapter 6.

Chapter 2

Related Work

2.1 Programming Frameworks for Wide-area Computing

Programming frameworks in a wide-area environment must handle complex tasks: enabling flexible interaction among processes, accommodating dynamic process joins and leaves, and overcoming network communication barriers. Moreover, these frameworks must not only accomplish this, but also do so in a way that requires minimal work and constraints on the framework user.

2.1.1 Flexible Inter-process Interactions

2.1.1.1 Grid-enabled batch scheduler frameworks

In order to address a large range of applications on the wide-area computing, programming frameworks need to provide a simple yet flexible means by which processes may interact. Grid-enabled batch schedulers [24, 62] has been used conventinally, to distribute work in clusters, and cluster of clusters environments. Users create self-contained script files which describe what he or she wants done on a compute node; this may be a shell script file, or a scheduler specific file format. The script files are submitted to the scheduler, and it determines on which node to distribute the task, based on its scheduling policy. This automatically distributes work for the user. However, this comes at a price that it is hard to express a distributed application that spans multiple such nodes; one may try to manually divide up the application into seperate scripts so that portions of the whole applications will be distributed across all nodes, yet there is no guarantee that they will be run concurrently. Some batch schedulers allow users to express a DAG inter-task dependency graph in a simple script file [2]. The scheduler automatically dispatches jobs whose dependencies have been satisified. Yet still, this does not address how tasks can pass data across each other, as in the case with parallel and distributed applications. For inter-task communication, tasks must output data to temporary files to passed to dependent tasks.

2.1.1.2 Master-worker frameworks

The master-worker model creates a hierarchy where a master process distributes work to numerous worker processes. The master takes a problem and divides it into subtasks to be distributed to worker nodes. The model takes a very simple communication pattern where the master first sends a subtask to a worker, and gets the results when the subtask is done. The master controls how to divide up the tasks and to which worker a job is distributed. It is an accepted paradigm for its simplicity [15]. As such, many frameworks specialize in this type of application [22, 46, 49, 59]. However, the model becomes complicated when the application requires frequent communication and interaction among nodes, as in the case with general distributed applications. The model only provides 2 types of communication: send a subtask, return the results. If the master and its workers require frequent interaction, the assigned tasks must be artificially broken into smaller subtasks. Some frameworks [22, 49] enable messages to be sent between the two parties to resolve this issue, but this often results in cluttered code.

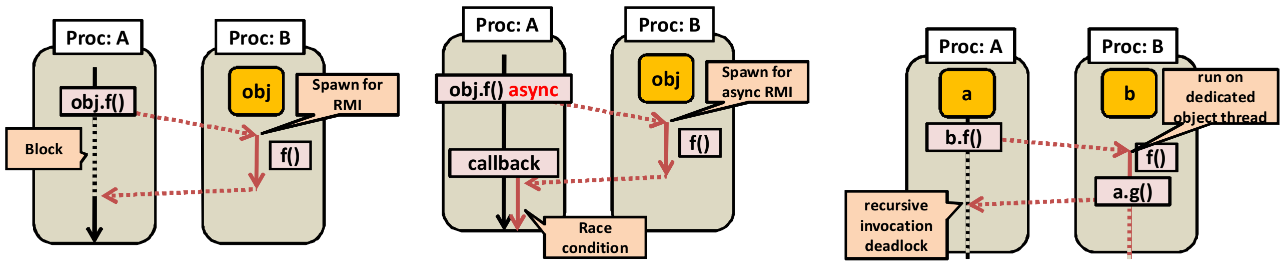
2.1.1.3 Divide-and-conquer frameworks

Satin [66] and distributed-Cilk [34] are frameworks for distributed divide-and-conquer computation, where a single large task can recursively be divided into smaller subtasks. The computation first initiates at one compute node where the single target problem is computed. The problem is *spawned*, or expanded into smaller subtasks. Idle compute nodes steal an un-attended subtask from another node, where it too recursively compute and spawn more subtasks. This effectively turns the target problem into a tree, where tree nodes correspond to subtasks. When a compute node steals an unattended subtask, it becomes responsible of working on the subtree of the problem rooted at the subtask. Eventually all compute nodes will obtain some subtasks and the work is automatically distributed. Completed subtasks are returned to the parent task, which is *syncing* or blocking for all children tasks to return. A parent task uses all children results to complete its task, and returns to its parent task. The advantage of this model is that a subtask corresponds to a single function call, and the user can program the applications by solely using recursive function calls, and the runtime automatically distributes the function call across computing nodes. However, this model's effectiveness is reserved to applications that can be expresed using recursive function calls (e.g.: calculating Fibonacci number, FFT, sorting, N-queen, etc). Other application types must somehow be modified to fit the spawn-sync recusive sub-dividing model, which is in itself hard and makes the application awkward.

2.1.1.4 Distributed object-oriented Models

This work argues in favor of distributed object-oriented programming models. These models extend on the general-purpose object-oriented languages where methods are invoked on objects for computation. The difference is that the model now introduces *distributed objects*. Distributed objects are dispersed among processes on different computing nodes. Methods may be invokes in them, as in the original programming language, and when an invocation on a remote distributed object occurs, the method and its arguments are serialized into a byte array and sent to the computing node which hosts the target object. There, the computation of the method is done and when it completes, the return notification and any results are again serialized and sent back to the invoker. It appears like a simple method invocation on an object but the implementation carries out the necessary communication transparently. This is shown in Figure 2.1(a). They are called RMIs (Remote Method Invocations), and serve as a good abstraction of communication and delegation of work to a remote compute node.

It is also possible to make RMIs *asynchronously* so that the invoker does not wait for their completion. The invoker by doing so, can make numerous RMIs to different objects concurrently, and



a. RMI flow b. Race-condition in RMI c. Deadlock with active objects

Figure 2.1: RMI outline

effectively express parallelism.

The advantage of this model is that it simply extends on general-purpose programming languages. Thus, it does not restrict the programmer to a particular programming style specific to parallel and distributed computing, and the transition from sequential programming is smoother. More importantly, it retains the programming flexibility of general-purpose programming languages.

2.1.2 Managing Parallelism for distributed objects

Extending an exisiting object oriented programming language for distributed computing is a popular approach. With Java, Ibis RMI [64] and ProActive [37], with Python, DisPyte [35] are notable examples. In these frameworks, parallel is expressed via asynchronous RMIs, where the invocation is made non-blockingly and a callback is invoked when the invocation results become available. However, as shown in Figure 2.1(b), RMIs can often result in race-conditions: 2 concurrent invocations may occur on the object, or the callback may be invoked concurrently with operations by the invoker. This makes it necessary to use mutual exclusion inside methods for distributed objects.

The active object [37] model takes an approach such that each object has a dedicated thread for execution. The advantage is that processing of all invocation requests are done sequentially and there can be no race-conditions. Yet this can easily induce deadlocks. This happens when the dedicated object thread blocks, either doing an synchronous RMI or waiting for results for a previously done asynchronous RMI. If the very invocation it has made calls back on the object, the dedicated thread is blocking and cannot handle the invocation. In such a case, no more progress can be made. An example is shown in Figure 2.1(c).

This work proposes a novel synchronization semantics for objects where *at most* 1 thread can operate on an object at any given time. When a thread performs an invocation on the object, it must first acquire its ownership. This implicitly achieves mutual exclusion. This itself is similar to active objects, but when the owner thread blocks in the object's context, the ownership is temporarily released. This prevents deadlocks that troubled active objects.

2.1.3 Handling process joins/failures with ease

On large-scale wide-area computing, where resources fluctuate constantly, it is paramount that handling of process joins and leaves is made simple for the user.

Satin [66], which utilizes the divide-and-conquer model, transparently performs load-balancing via Random Work-Stealing [26], where idle compute nodes steal unattended tasks from randomly selected other nodes. In these divide-and-conquer programming model where each resursively called function compose a subtask, each compute node computes subtasks in a depth-first manner just as calling functions in any procedural language. Thus, many subtasks closer to the root in the task tree are left unattended. Because tasks closer to the root are larger tasks (i.e.: it has a larger task subtree), and because these tasks will be stolen by other nodes, Random Work-Stealing performs analytically efficient load-balancing. This method effectively makes node joins trivial as that node only has to steal an idled task. Satin also transparently handles node failures as well. When a compute node fails, it is assumed that the task tree rooted at that compute node is lost, and lost tasks are simply re-executed on a different node. The advantage of applications that can be expressed in the divide-and-conquer model is that computing a subtask does not have any side-effects on other tasks, and re-execution is trivial. Though the divide-and-conquer model can handle node joins and failures effectively, its methods are specific to the model and is very difficult to apply it to other models.

Master-worker frameworks like Jojo2 [22] handle worker joins/leaves via event handlers. The user implements handlers that will be invoked on the master object in case of such events. Yet, node joins and failures are asynchronous events from the master, and in order to avoid race-conditions, mutual exclusion is essential. It is possible to avoid this if the master node's code is modified into an event-driven loop; a single master thread unblocks on each event (e.g.: node joins, node failures, task job completion) to execute corresponding operations (e.g.: distributing new tasks, re-enqueuing failed tasks). This is very simple for a generic master-work application, but if a general parallel and distributed application were to be implemented in this fashion, the code will quickly become hard to understand, not the mention the required synchronization.

This paper's proposal presents, in conjunction with the new synchronization semantics, a signaling mechanism for each object, which unblocks an arbitrary thread blocking in the object's context. When a thread is unblocked and starts running after it gains exclusive access to the object, it can handle operations necessary for the node join. Thus it is possible to handle asynchronous events such as node joins while adhering to the proposed implicit synchronization semantics. Additionally, process failures are abstracted as exceptions to method invocations. These exceptions can be caught as in any other generic exceptions, and the procedures to handle node failures can be written there. Thus, this widely accepted semantics nicely integrates failures into conventional programming. The contribution of these constructs is that handling node joins and leaves can be implemented in code without disrupting the main processing code of the application.

2.1.4 Resolving connectivity on dynamic wide-area networks

Realizing communication on WANs, where connectivity is limited by NATs and firewalls, in a scalable manner is a difficult task. Communication scalability is particularly crucial on large-scale environments when resource is limited. Such resources may be file descriptor limits and memory usage at compute nodes, and/or state information at intermediate switches and routers. Furthermore, the overlay must tolerate dynamic changes to the network as compute nodes join and leave. Yet in resolving these issues, programming frameworks must only require minimal configuration burden and restrictions on the user.

PadicoTM [32] enables distributed computing using CORBA on various network hardware, yet does not take connectivity issues into account. ProActive [37] requires users to hand write descriptor files that specify connectable points. Realistically however, the programmer cannot specify how all connections are established when the scale of the computation is large, and in a dynamic environment connectable points change on a day to day basis. This is a high burden on the user. Jojo2 [22] combines SSH to reach all cluster head-nodes, and UDP broadcast among intra-cluter nodes to create a 2-level hierarchy to achieve connectivity. Yet the system requires users to configure all individual cluster head nodes. SmartSockets [47] attempts to establish connectivity on WANs by transparently attempting various methods to establish TCP connection(s), but requires users to setup router nodes manually. Moreover, because it will establish direct socket connections for all directly connectable points, this does not address scalability issues. None of the above except [22] take into account of connection scalability and dynamic changes to the network.

This work proposes to construct an overlay network over which communication is routed transparently. Each processes establishes a small number of connections, and address the connection scalability issue. Nodes may join and leave the overlay during runtime, also long as the overlay does not become partitioned, and the routing adapts to the changes. By default it does not require any user configuration and, the overlay construction scheme at each node chooses random peer nodes to yield a connected overlay graph by a high probability. For special cases like firewalls and multiple private networks/-NATs, the user can concisely direct the framework to modify peer selection and/or connection type (raw or SSH-forwarded TCP connection).

2.2 Application Overlays for Wide-area Distributed Computing

Applications overlays for parallel and distributed computing, in addition to providing reliable communication, must be able to deliver performance comparable to raw sockets with respect to both latency and throughput. Communication latency on wide-area overlays is mainly dictated by what kind of overlay graph is constructed and how point-point communication is routed; data must not take roundabout hops on the overlay for minimal latency overhead. Throughput, on top of the above, is also governed by the flow-control and buffer management scheme on the overlay; while it is important that data is forwarded as fast as possible for maximum throughput, it must not unreasonably overload the intermediate nodes or cause memory overflow. Not only must overlays create an underlying network aware graph and paths, but also implement an efficient flow-controlling algorithm.

2.2.1 Existing overlays that deploy on wide-area networks

There have been research to improve communication performance and reliability in WAN environments [45, 51], but the implementation of flow control have been secondary. RON [19] is an overlay architecture that establish UDP links among all nodes on the overlay. Its aim is not to resolve connectivity, but to give resilience to internet path outages and better communication performance. Even if there is a physical outage on the path taken by a direct IP-layer path, RON can use multiple link hops to route around the problem. Additionally in WANs, there are even cases where the direct internet path does not give optimal performance, both in latency and bandwidth. RON overlay nodes monitor the performance of all links and routes packets so as to maximize a certain metric; the direct link is not necessarily taken depending on its quality compared to multi-link paths. RON however only exposes a UDP-like API to the user, so reliability and flow control is delegated to the user.

DiskRouter [41] is a data transfer overlay across WANs, whose primary objective is a file-transfer middleware between distributed tasks in the Condor batch scheduler. It can provide multiple overlay paths between the source and destination via intermediate DiskRouters, and provides resilience to path outage and performance boosts by striping the data across all paths. In its simplest form, a DiskRouter is a store-and-forward device and receives and buffers data locally until it can be sent on the next link. This can potentially require a very large buffer space, especially in heteroeneous environments, and so it uses local memory *and* disk to buffer all data. However, unlimited buffer is unacceptable for performance and load on DiskRouters, so it is possible to set a pre-defined threshold for buffers. If a threshold is reached for the buffer, it may be configured to back-pressure its predecessor. The back-pressure among multiple transfers can cause the overlay communication to deadlock, if multiple transfers co-exist.

2.2.2 Flow control protocols for overlays

Some research that aim at designing flow control for overlays have the system provide explicit congestion feedback to the sender so as to adjust its send rate [16, 38, 43]. In these works, the sender receives notifications as to any congestions in links that it is using. Kar et al. [38] proposed to have the number of congested links returned to the sender in the form of ACK packets. The sender then adjusts its send rate accordingly.

Amir et al. [16] proposed a global flow control scheme where all overlay nodes are updated on congestion status of all links in the network. According to how much data is pending in a link's user-level send buffer, a "cost" is assigned to it. Each node is given a predefined amount of credits per epoch, and it can only send packets along a path when it can buy up costs of all the links it uses. Otherwise, the node has to wait until it accumulates enough credits. This scheme realizes flow control using a demand and supply feedback for the link's bandwidth capacity. However this is unrealistic in a large-scale wide-area environment as it would requireperiodic broadcast among all nodes.

Experiments were conducted with Spines [17], an available implementation of [16] in a Gigabit-Ethernet LAN. However, the overlay was only able to utilize 300[Mbps] throughput for point-point transfer. This experiment demonstrated for us that in reality, it is difficult to have these protocols fully adapt to the network environment at hand. This work does away with explicit congestion notifications by taking advantage of TCP. Using TCP makes maximizing throughput utilization trivial, as so much work has already gone into TCP for this purpose. Flow-control is implemented by back-pressure between dependent connections in a path.

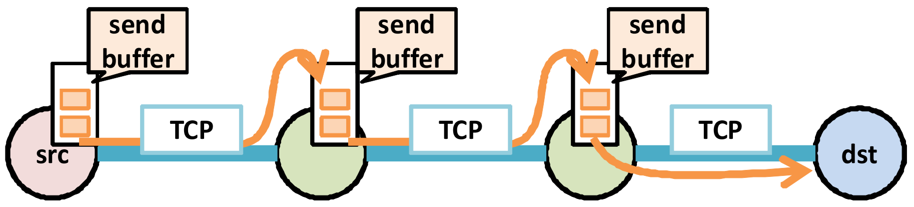
2.2.3 TCP and fixed-buffer coupling for peer-to-peer multicast

It is also possible to implement flow control across multiple TCP connections by coupling TCP connections and fixed intermediate buffers. In this method, as shown in Figure 2.2(a), each node reserves a fixed user-level send buffer queue per TCP connection. Any received data packet that needs to be forwarded is immediately appended to the send buffer queue of the next TCP connection. If this send queue becomes full due to congestion, the node will stop receiving packets from the preceding connection. TCP will also block the send on the preceding node and back-propagate the congestion implicitly. This method is outlined in Figure 2.2(b). The advantage of this method is that it requires no overhead other than the flow-control scheme by TCP, and can be implemented very efficiently. Some multicast overlays adopt this mechanism to achieve flow control from the root to leaf nodes [44, 63]. However, unlike in multicasts where there exists only one pre-known flow, naively adopting this on overlays for general unknown traffic pattern will result in a deadlock. This work resolves this problem by adopting deadlock-free routing.

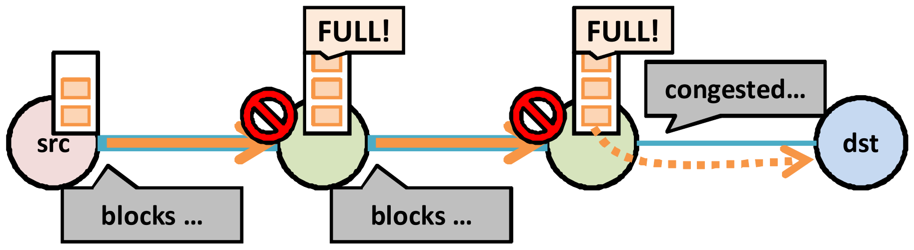
2.2.4 Deadlock-free routing

Deadlock-free routing algorithms [21, 29, 30] have been proposed for communication networks for message passing concurrent computers. These algorithms prevent deadlocks for switches that route packets between computing nodes, by imposing restrictions on the used path routes. In particular, they ensure that there can be no cycles in the link dependency graph. Many previous work focused on deadlock-free routing algorithms for high-performance interconnects with specific topologies like k-ary n-cubes (e.g.: hypercubes and tori).

More recent algorithms [28, 40, 57] do not assume any form of network topology and prevent deadlock on arbitrary networks. This approach is applicable to achieving flow control by coupling TCP and fixed-buffers without causing deadlocks. However, current algorithms do not account for heterogeneity on WANs, and they can impose deadlock-free contraints with large performance degradations. For example, 2 intra-cluster node pairs may be forced to use inter-cluster wide-area overlay links simply because of poor constraints. This work proposes heuristics that adapt deadlock-free algorithms to such



a. Flow-control overview



b. Congestion backpropagation

Figure 2.2: Flow-control by coupling TCP and fixed buffers

networks. It also shows how its routing table can be computed distributedly by extending Dijkstra's shortest-path algorithm.

2.2.5 Utilizing underlay information for overlay

There are also previous work that incorporate the underlying topology information to improve the overlay performance. RON [19] creates a complete graph of all overlay nodes, and probes each link periodically to dynamically decide which path to take. In certain instances, a roundabout path may yield better performance than a direct one. However, the all-to-all probing hinders scalability. PLUTO [50] is an underlay information query service on PlanetLab [9]. The service probes the L3 information of the network and presents AS-level connectivity information to the user, which can be used to plan the overlay construction and paths [51]. Though it is highly useful, it is not available beyond PlanetLab. This work, to improve the quality of the overlay, utilizes the latency matrix among peers and bandwidth values for TCP connections established. The bandwidth values can be computed at run-time, and [55] shows a way to create approximate latency matrices only using few measurements, thus all necessary information can easily be obtained by the user.

2.2.6 Performance-aware connection management for overlays

Besides NAT/firewall concerns, it is not desirable to aggressively establish connections, due to the large number of nodes in large-scale wide-area environments. TCP connections, especially wide-area connections, consume large amounts of memory per host because large enough send/receive buffers have to be reserved to meet the bandwidth-delay product. In addition, WANs introduce many bottleneck links and creating fewer connections does not reduce performance by large factors. In fact, aggressively communicating on many WAN sockets concurrently is even harmful, and can even reduce performance by inducing large amounts of packet-loss. With this in mind, it is possible to reduce the number of connections in the overlay, while still retaining communication performance on the overlay. In an MPI implementation by Kielmann et al. [39], a gateway node is selected per cluster, and connections are established among all gateway nodes: all inter-cluster communication goes through gateway nodes. However, this can be unnecessarily prohibitive and introduces bottlenecks at gateway nodes. Additionally, with the rise of 10[Gbps] WAN backbones, a single 1[Gbps] connection cannot saturate the available bandwidth. Saito et al. [55] proposed to create a locality-aware connection scheme where each node establishes connections to closer nodes densely, while spaserly to far-away nodes. This heuristics realizes scalability bounding the order of connections to $O(N \log N)$ from $O(N^2)$, where N denotes the number of nodes, while minimizing communication overhead to closer nodes sensitive to performance. This work will show that the proposed overlay can be improved by employing this. Young et al. [68] proposed to establish connections along k interleaved spanning trees, where k is a user defined constant. Effectively, this will establish more connections among close nodes. This in spirit is similar to what this work intends to do.

Chapter 3

gluepy: A Distributed Programming Framework for Wide-area Environments

This chapter presents the model and the implementation of the proposed programming framework for large-scale wide-area parallel and distributed computing. The framework addresses the complexities of large-scale wide-area environments and allows parallel and distributed application to be implemented straightforwardly using a general-purpose programming language.

3.1 The Framework Programming Model

This work proposes a distributed Python object-oriented framework that operates in a NAT/firewall-prone environment with dynamically joining/leaving nodes. To the user, it provides a seamless view of the underlying environment, and present a set of simple interfaces by which nodes communicate via RMIs, new nodes may join, and failing nodes are detected. Like other distributed object-oriented frameworks, the proposed model provides remote objects, and communication among them are abstracted in the form of RMIs [37, 64]. However, it addresses a number of topics important to Grid-enabled programming that were not, or only partially discussed before.

3.1.1 Synchronization and asynchronous event handling

3.1.1.1 Motivation for the model

In the context of parallel computing, parallel and asynchronous RMIs are crucial, as they are good abstractions for distributing computation among nodes. Yet, manipulating asynchronous RMIs is still a non-trivial task. Some implementations allow the users to define callbacks for when the results are available. However, this requires using locks to handle critical data. To resolve this issue, there are future primitives that allow the invoking thread to block for results when they become necessary; the invoking thread may perform other computation in the mean time. Its advantage is that a single thread is in control of the entire flow, and the transition from sequential programming is natural.

This does not resolve the issue on the RMI handler's standpoint. Since a remote object may receive an incoming invocation handled by an independent thread at any time, the programmer must use locks for objects that *might* receive incoming RMIs. To resolve this problem, some models [37] have implemented *active objects* where there is a dedicated thread for each object. The dedicated thread handles all incoming RMIs sequentially. However, this model easily creates deadlocks when an RMI handler also is an RMI invoker.

Yet another issue arises when the program has to handle asynchronous events, such as new node joins. The RMI-based model alone cannot handle these events. One possible approach is to handle RMI callbacks, node failures, and node joins all in one single event driven loop, like in many other master-worker frameworks. The obvious advantage is that a single control thread does all operations, eliminating locks and conditional variables. However the programmer must take care so that event handlers do not block, and the natural flow derived from sequential programming is completely lost.

The qualities favorable in a distributed object-oriented model is summarized below.

- provides future primitives for expressing parallelism
- allows objects to be accessed mutually exclusively without explicit locks
- avoids unpleasant deadlocks induced by implicit serialization semantics above
- may handle asynchronous events without low-level event handling models

3.1.1.2 Model and API

In the proposed model, at most one thread may run on an object at a time. For a thread to invoke a method of an object, it must first implicitly acquire the object's *ownership*. The thread keeps the ownership for the duration of the method invocation, and releases it when the invocation ends. This is equivalent to using a lock to serialize accesses to a critical region, and accesses to an object by multiple threads are serialized as in Figure 3.1(a). While in the scope of the method, it is possible for the owner thread to *block*, as defined below:

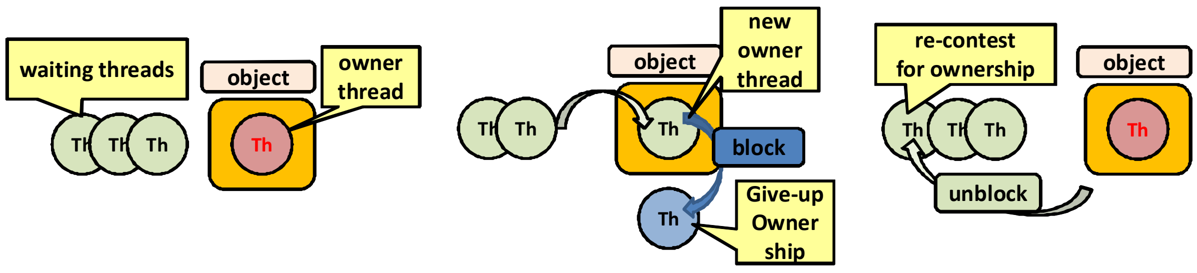
- a synchronous object invocation
- waiting for the results of an asynchronous object invocation

When the owner thread blocks, it temporarily releases the ownership and another pending thread is permitted to run, as shown in Figure 3.1(b). When there are more than one pending thread, an arbitrary thread is scheduled, and acquires the ownership. After the blocked thread unblocks, it joins the group of pending threads waiting on the object as in Figure 3.1(c), and resumes execution only after re-acquiring the ownership. The model supplies future primitives by which threads may block for results. Finally, it provides a signaling mechanism for each object, by which a thread blocking on future primitives in the object's context is made to unblock.

Specific APIs for asynchronous invocations and synchronization is as follows. When an asynchronous RMI is performed, the invocation returns immediately with a future object. To do an RMI `fib` on an object `foo`, do the following.

```
future = foo.fib.future(args)
```

In order to obtain the results for the asynchronous RMI, do



a. Access serialization b. Ownership release c. Ownership re-acquire

Figure 3.1: Proposed serialization semantics

```
result = future.get()
```

If the results are not available, the call will block until they are. For scheduling, it is also very common that one waits for an array of future objects simultaneously, until any result becomes ready. To this end, the model provides a global `wait` primitive that takes a list of future objects, blocks until at least one of the futures' results are available, and returns a list of futures whose results are available.

```
ready = wait(futures)
```

Finally, each object is provided with a signaling primitive invoked by

```
obj.signal()
```

This forcefully unblocks a thread that is blocking on `wait` in the object's context. If no threads are blocking at that time, the *next* thread that calls on `wait` in its context will be woken. The woken thread will contest for and reaquire the object ownership, after which the unblocked `wait` primitive will return None.

3.1.1.3 Model rationale

This serialization semantics eliminates the need for explicit locking. This is similar to that of active objects, adopted in some object-based languages [37]. Active objects, however, easily lead to an unpleasant behavior called self-recursion deadlocks. In contrast, the proposed model allows another thread to run on an object when the current thread blocks in the midst of a method execution (a synchronous RMI, call on `wait`, or `future.get`). This property prevents deadlocks due to such recursive calls. With respect to atomicity, a thread is guaranteed to have exclusive control in between potentially blocking operations. Thus, the object's state may be modified without worrying about races conditions.

Furthermore, a special method `signal` allows to unblock a thread currently waiting on the object. This is similar to the semantics of UNIX signals, which unblocks threads blocking on some I/O system calls (e.g., read). This can be used to wake a blocking thread for handling of some event(s).

3.1.2 Failure semantics and bootstrapping nodes

For RMI failures and object lookups, the model utilize the semantics widely accepted in existing RMI frameworks. Node failures are commonplace on large-scale environments, and simple means for handling them must be presented to the programmer. To this end, node failures are abstracted as exceptions for RMIs to objects on failed nodes. The user may catch such exceptions for failure handling. Aside from uncaught application-level exceptions in an RMI, when any of the below failures occur during an RMI, a `RemoteException` is raised on the caller side. In all cases, they may be handled by the invoker to perform recovery or evasive measures in the regular Python semantics.

1. The communication route between the caller and the callee nodes is broken
2. The callee node fails during execution

Another crucial issue for wide-area computing with dynamically joining nodes is bootstrapping a node. In distributed object-oriented models, this translates to obtaining the *first reference to a remote object*. In the proposed framework, any remote object may be *published* with a human readable string name as follows:

```
object.register("any-name")
```

A process may obtain a reference to a published object using the name.

```
ref = RemoteRef("any-name")
```

This way, the newly joining node may obtain the reference to an existing object and thus join the computation by notifying existing members of its joining.

3.1.3 Sample code

Using the proposed programming model, one can easily implement applications using dynamic processes. One of such examples is shown in Figure 3.2(a), a simple yet complete template for master-worker applications. The master initiates work on each worker via asynchronous RMIs, and results are collected using futures. The `nodeJoin` method, invoked by workers, uses the `signal` primitive to unblock the master and add new workers into the workers list. Node failures are handled by catching exceptions for respective RMIs, and re-submitting the work. It is noteworthy that locks are not necessary, and that by using futures, the master's flow resembles that of a sequential program. With the active object model, where locks are also unnecessary, maintaining this flow is impossible as the master object is in method `run` the entire time; workers will never have a chance to run `nodeJoin`. In comparsion, Figure 3.2(b) shows the code for a typical master-worker framework. A handler, independently invoked on each event, has to branch of each event type, and the loop must perform explicit mutual exclusion. The event-driven code also destroys natural sequential flow of control.

3.1.4 Discussion

In the proposed object semantics, atomic blocks do not encompass an entire method block, but rather between potentially blocking operations within a method. Yet this is acceptable semantics for the following reasons. Atomic sections in real life applications are very short (e.g., checking if a given value exists in a map before insertion). Moreover, users are aware of blocking operations in advance (synchronous RMIs, access to futures, calling `wait`). Also, as common practice, it is not favorable to design atomic blocks such that they encompass blocking operations.

```
class Master:
  def __init__(self, jobs):
    self.nodes = []
    self.jobs = jobs

  def nodeJoin(self, node):
    self.nodes.append(node)
    self.signal() # notify join

  def run(self):
    assigned = {}
    while True:
      # dispatch work to available workers
      while len(self.nodes)>0
            and len(self.jobs)>0:
        node = self.nodes.pop()
        job = self.jobs.pop()
        # asynchrous RMI to worker
        f = node.work.future(job)
        assigned[f] = (node, job)

      # wait for any results
      readys = wait(assigned.keys())

      # if got signal, loop back
      if readys == None: continue

      #read ready results
      for f in readys:
        node, job = assigned.pop(f)
        try:
          print "done:", f.get()
          self.nodes.append(node)
        except RemoteException, e:
          # in case of a fault, rerun job
          self.jobs.append(job)

class Worker:
  def work(self, job):
    #do work on job...
    return results

  def run(self, mastername):
    #obtain reference to master and join
    master = RemoteRef(mastername)
    master.nodeJoin(self)
```

a. The core for a simple Master-Worker Program. Classes for the Master and the Worker are shown.

```
class Master:
  def __init__(self, jobs):
      self.jobs = jobs
      self.workers = []
      self.tabs = {}
      self.lock = Lock() # for mutex

  #invoked on new event with arg. e
  def handleEvent(self, e):
      #need mutual exclusion
      self.lock.acquire()
      try:
        #give job to new node
        if e.type == NEW_NODE:
            node = e.node
            #give new job, if any
            if len(self.jobs) > 0:
                job = self.jobs.pop()
                self.tabs[node] = job
                self.giveJob(node, job)
            else:
                self.workers.append(node)

        #handle result and give-out a new job
        elif e.type == JOB_DONE:
            print "done:", e.result
            node = e.node
            #give new job, if any
            if len(self.jobs) > 0:
                job = self.jobs.pop()
                self.tabs[node] = job
                self.giveJob(node, job)
            else:
                self.workers.append(node)

        #re-enque lost job on node failure
        #do not re-enqueue worker
        elif e.type == FAILURE:
            node = e.node
            job = self.tabs.pop(node)
            self.jobs.append(job)
      finally:
        self.lock.release()
```

b. Master-Program template in a typical master-worker framework

Figure 3.2: Sample code and comparison with typical master-worker framework

In the semantics however, livelocks may still occur, like in cases where a thread infinitely loops in a method without blocking. This is arguably as hard to debug as deadlocks. Currently, this is deferred as future work.

The proposed signal mechanism sends the signal to *objects* rather than to *threads*. This design decision was motivated by the fact that in a distributed non-active object model, threads are ephemeral existences only used to gain parallelism. Thus, the programmer is more concerned about interacting with objects, than threads.

Finally, the proposed model can address a wide range of applications beyond the master-worker model shown in the sample code. For example, more P2P-like applications like distributed island-GA applications, where each node performs GA and periodically do crossovers with other peer nodes, can also be easily expressed using RMIs to each other's object. Yet another example is a P2P application that compute PageRank for large number of documents in a parallel and distributed fashion [53]. In this application, each compute node exchanges PageRank information for locally stored documents with random peers to improve the accuracy of the overall scores. Such P2P applications are on the rise, can be naturally expressed using the general distributed object-oriented model as in the proposed framework.

```
# for all peers whose IP address pattern matches <src-pat>,
# connections to peers whose IP address match <dst-pat> will
# use SSH-portforwarding with SSH user <user>

require   <src-pat>  <dst-pat>,   prot = ssh,   user = <ssh-login-user>
```

Figure 3.3: gluepy connection configuration file syntax

3.2 The Framework Implementation

The following section discusses how the framework implementation copes with the physical issues of large-scale wide-area environments. In particular, the implementation had to resolve three issues: point-to-point communication in a WAN setting (NATs, firewalls), allowing nodes to join with ease, tolerating abrupt node failures.

3.2.1 Overlay network construction

The framework, to realize point-to-point communication among all nodes, automatically construct an overlay using TCP connections. Each participating node establishes connections with a small number of nodes chosen at random (about 20 connections). Analysis has shown that such a scheme will create a connected graph of all participating nodes with high probability [36]. However, some cluster completely filter incoming and outgoing packets, and thus there are no means by which these resources may be connected. For these cases, the scheme automatically perform SSH portforwarding over which TCP connections are forcefully established. Only for these cases, the scheme requires the user to specify the points between which SSH portforwarding is done. However, adjusting the scheme to use portforwarded connections between 2 groups of nodes only requires one line in a configuration file as in Figure 3.3.

Over this overlay, a routing layer adapted from a reactive routing protocol, AODV [54] is implemented. It adapts well to dynamic graph changes and fits the setting where nodes join and leave at will; the lazy routing path construction does not entail broadcast storms in face of high churn. The routing metric is the RTT latency for each TCP link.

3.2.2 Dynamic node join

For nodes to join the computation, it must first become connected with the overlay. In order to do so, it needs a set of bootstrap peer node information, or *endpoints*, with which it will first connect. In a TCP overlay, this entails obtaining a set of initial (IP, port) pairs. The framework thus provides an *endpoint server* that all nodes access before joining. Each node obtains a set of endpoints. It then adds its *own* endpoint to the server so that other nodes may connect to itself. A number of options are available for this server. One is an HTTP server, the other is a server built on top of GXP [1][60], a Grid shell. Using GXP, one may log into hundreds of remote servers via SSH and execute commands

[1]http://www.logos.ic.i.u-tokyo.ac.jp/gxp/

on them in parallel. It also provides a mechanism with which all nodes may communicate via SSH tunnels. Because all communication is done via SSH, this mechanism can be used even for resources that are not accessible by any other means.

3.2.3 RMI fault detection

In the context of an overlay network, a communication route between 2 points may constitute more than 1 TCP connection, and thus is not trivial to detect RMI faults. It is assumed that when a node fails, it closes all established TCP connections. An RMI is realized by two protocol messages, the RMI Request and the RMI Return message. Additionally, each RMI is identified by a globally unique id, an *RMIID*. In the implementation, an RMI failure is defined as when *either the RMI handler node fails, or any of the TCP connections that the RMI Request message has traversed fails.*

3.2.3.1 RMI Invocation and Return Implementation

First, the process of an RMI invocation is shown in Figure 3.4(a). On method invocation, an RMI Request Message is sent towards the object hosting node. Each RMI Request Message contains the following.

RMIID: globally unique ID for the RMI

path: list of nodes it has visited, starting from the invoking node

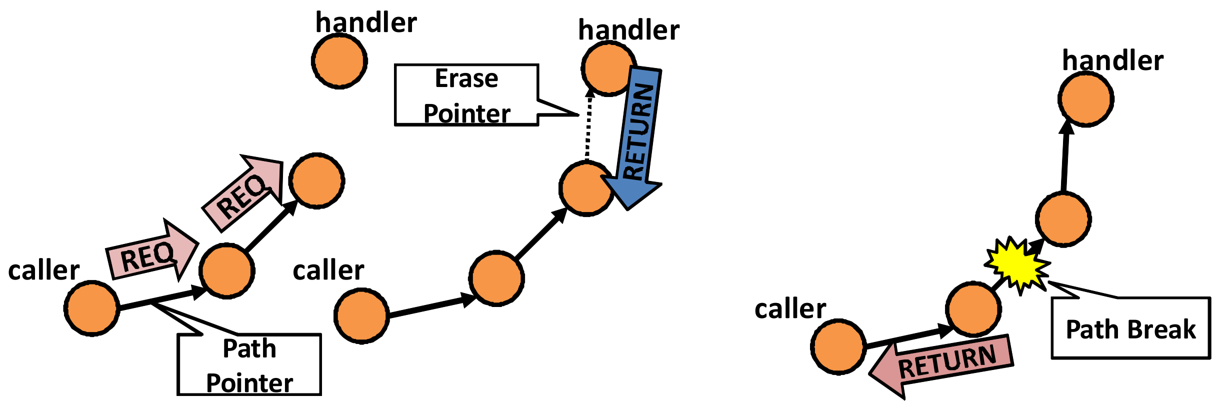
Before a node sends/forwards the message, it does the following operation.

1. using the RMIID as the key, locally store the path traveled by the RMI Request Message as the return path
2. using the RMIID as the key, locally store a *path pointer*, which is a pointer that points to the connection to which the message will be forwarded
3. add itself to the path in the RMI Request Message

On the other hand, after the method invocation has been handled by the object hosting node, an RMI Return Message is returned towards the invoker. The message along with the RMIID is forwarded along the path on which the RMI Request Message came. As a node receives the message, it does the following operation.

1. remove the locally stored return path and path pointer for the RMIID
2. return the message along the path, if a connection to the next hop exists
3. if no connection to the next hop exists, simply drop the message

As it follows the path in reverse, all intermediate nodes erase the path pointer for the RMIID. If the node to which the message has to be returned does not exist due to node failure, the RMI Return Message can be ignored because of the clean-up operation triggered by that node failure. This is described below.



a. RMI Request and Return Message

b. clean up for RMI fault

Figure 3.4: RMI messaging implementation

3.2.3.2 Clean-up operation in case of intermediate node failure

When a node fails, all nodes connected to it will detect a connection failure. Each neighbor node finds out if any RMI path pointer points to the failed connection. For all such existing pointers, the node deletes the pointer and sends an RMI Return Message carrying a failure exception, back along the stored path. If the node to which the message must be sent is unavailable due to yet another node failure, the message can simply be ignored again. This is because the clean-up operation would have been triggered by this node failure as well. As long as the RMI invoker node has not failed, it is guaranteed to receive an RMI Return Message carrying either the RMI returns, or an RMI failure status. By doing so, the intermediate node can notify the RMI invoker of the RMI failure, and the path created by the RMI Request message is effectively torn down. An image of the entire process is shown in Figure 3.4(b).

Chapter 4

Deadlock-free Overlay for Wide-area Networks

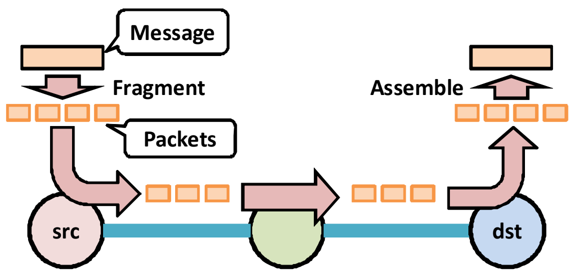
This chapter presents the proposed wide-area overlay that performs deadlock-free routing for all point-point communication in a locality-aware manner. The overlay can be used as the underlying communication layer for parallel and distributed programming framworks, so they may be deployed in wide-area environments to resolve connectivity and scalability concerns.

4.1 Overlay Overview

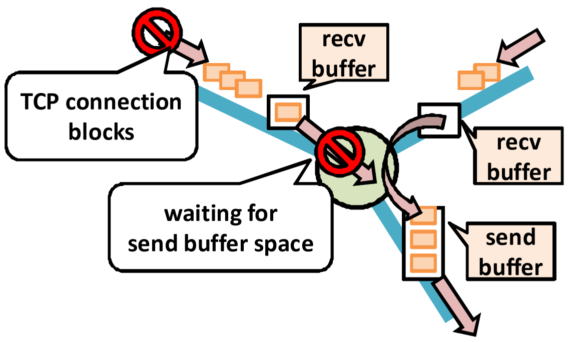
Our overlay system consists of 2 components: overlay construction, and deadlock-free routing table construction. The system first constructs an arbitrary TCP overlay, where each participating process establishes TCP connections to an arbitrary set of other participating processes. Using this graph of TCP connections, a deadlock-free routing table is computed distributedly. An arbitrary routing metric can be chosen for the computation: latency, hops, etc. Afterwards, point-to-point messages between processes are simply forwarded via intermediate processes, each using fixed buffer memory. The overlay guarantees reliability, and FIFO for all overlay communication while achieving flow control. It also assures reliability for the overlay *nodes* in the sense that there can be no memory overflow. In return, the overlay makes very few assumptions; all that is required is that the TCP overlay be a connected graph.

4.2 Forwarding Procedure on Overlay Nodes

This section describes the overlay node I/O procedure. All point-point communication is conducted in the form of *messages*. A messages corresponds to a continuous string of characters submitted by the application to be sent to a destination. Before transmission, each message is segmented into packets of predefined lengths, and they are sent in FIFO. The order of packets within a message is preserved while packets are forwarded to the destination, and no packets are dropped by intermediate overlay nodes. The packets are reassembled into one continuous message at the destination node before delivery. A graphical overview of a message transmission is shown in Figure 4.1(a)



a. Message transmission



b. Procedure at each node

Figure 4.1: Graphical representation of the forwarding procedure

A node performs all I/O operations non-blockingly. The set of all connections for a node refers to all the established TCP connections the node has. For each connection c, the following are defined.

- 1-packet receive buffer
- finite send buffer
- connection wait queue

Because receives are carried out non-blockingly, bytes received from a connection may only constitute a partial packet. Such a partial packet is temporarily stored in the connection's 1-packet receive buffer. Once a partial packet becomes a full packet after subsequent receives, it will be removed and be either delivered to the user, or forwarded to the next connection. A full packet also remains in this buffer if the next connection's send buffer is full; in such a case receives on the connection is blocked.

Each connection has a finite send buffer, where packets are sent in FIFO. Sends are done non-blockingly, and packets that could not be fully sent are returned to the head of the send buffer. If the send buffer is full, a packet from another connection will fail to append it onto the buffer. In such a case, the failed connection will enter the wait-queue of the connection, and receives on it will stop.

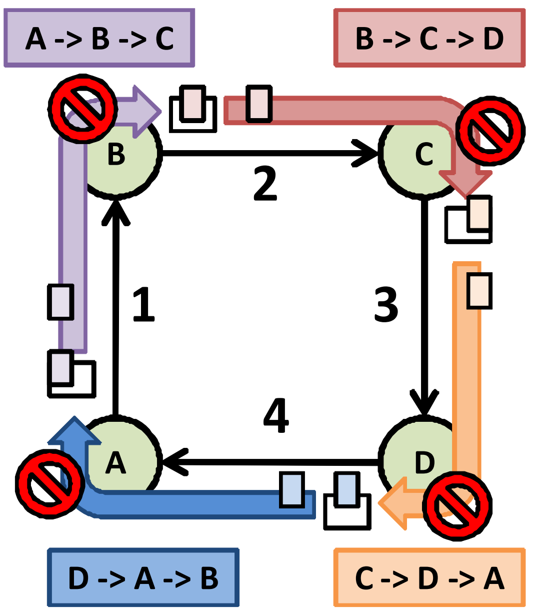
A connection's wait-queue is a FIFO queue of connections waiting for the its send-buffer to become non-full. Whenever a packet is removed and full sent from its send-buffer, the connection's wait-queue is checked and any pending connections are popped and its pending packet is appended on the send queue. Once a connection is removed from a wait-queue, receives on it will resume.

The overlay node follows the I/O procedure shown in Figure 4.2. In the figure, packets that could not be fully sent non-blockingly by `send` is returned. The overview of the procedure at each node is graphically shown in Figure 4.1(b) The procedure effectively implements the back-pressure mechanism using TCP connections and fixed buffer queues. The system does not deadlock thanks to deadlock-free routing.

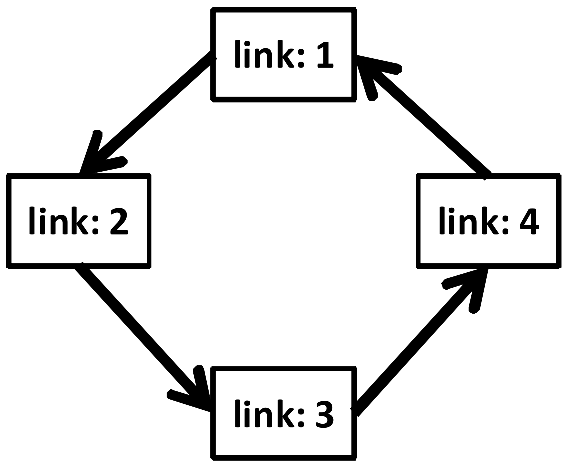
attributes for each connection c**:**
c.buffer: queue of strings to be send on connection c
c.waitqueue: queue of connections waiting on connection c
c.packet: partial packet from connection c, initially empty string

```
io_loop() {
  $RS$ = set of all connections;
  $WS = \{\}$;
  while ( true ){
    $R, W$ = block and get set of readable/writable
             connections from $RS$ and $WS$;
    foreach ( $c \in R$ ){
      $c.packet = c.packet$ + recv packet from $c$;
      if ( $c.packet$ is partial ) continue;
      if ( destination of $c.packet$ is here ){
        deliver $c.packet$ to user;
      }else{ /* if need to forward */
        $n$ = get next hop connection for $c.packet$;
        if ( $n.buffer$ is full ){ /* $c$ needs to wait */
          $n.waitqueue.enqueue(c)$;
          $RS = RS \setminus \{c\}$; continue;
        }else{ /* pass packet to next send buffer */
          $n.buffer.enqueue(c.packet)$;
          $WS = WS \cup \{n\}$;
        }
      }
      $c.packet$ = ""; /* reset receiving packet */
    }

    foreach ( $c \in W$ ){
      $p = c.buffer.dequeue()$; /* get a packet to send */
      $p'$ = send $p$ on $c$; /* get remainder in $p'$*/
      if ( $p' \neq$ "" ){ /* return remainder to queue */
        $c.buffer.enqueue\_head(p')$; continue;
      }

      /* waiting connections pass-on its packet and become active again */
      while ( $c.waitqueue$ is not empty and
              $c.buffer$ is not full ){
        $w = c.waitqueue.dequeue()$;
        $c.buffer.enqueue(w.packet)$; $w.packet$ = "";
        $RS = RS \cup \{w\}$; /* re-activate the waiter */
      }
      if($c.buffer$ is empty){$WS = WS \setminus \{c\}$}
    }
  }
}
```

Figure 4.2: I/O thread procedure



a. Deadlock image



b. Link dependency cycle

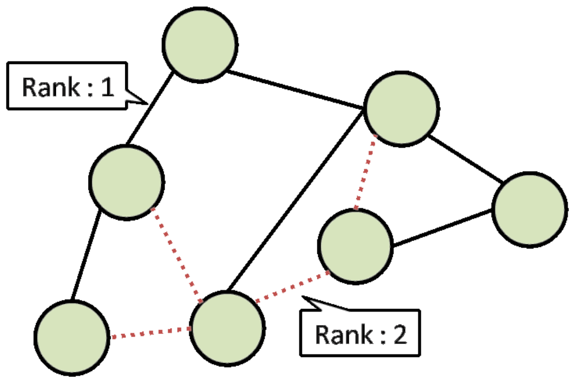
Figure 4.3: An example of 4 transfers in a deadlock

4.3 Deadlock-free Routing

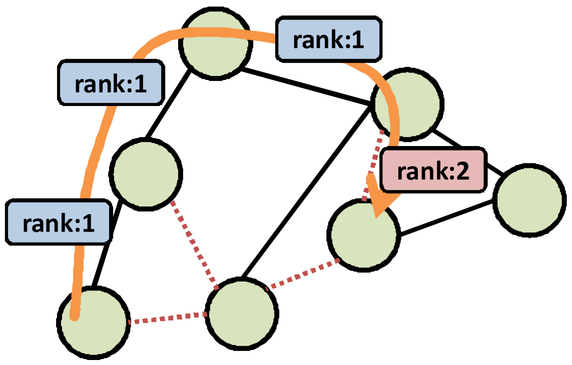
4.3.1 Deadlocks in computer networks

This section describes how a deadlock can occur when each node has a finite buffer. First, a simple example of a deadlock is presented. In an undirected graph, such as a TCP overlay, a finite send buffer is allocated for each link. Packets must be copied here before being sent on the link. If a node needs to forward a packet to a different node, the packet must be copied to a corresponding send buffer. If the send buffer is full, the node cannot receive the packet from the preceding link, since no buffer space is available. Consider an example scenario of 4 nodes in Figure 4.3(a) and let each link have a 1 packet send buffer. If any overlay paths have loops, then it is trivial that a deadlock can occur. Consider 4 transfers, $A \to B \to C$, $B \to C \to D$, $C \to D \to A$, $D \to A \to B$ that yield shortest paths. If they happen concurrently, each transfer will fill the send buffer for links $A \to B$, $B \to C$, $C \to D$, $D \to A$ respectively. However the next node will be unable to receive any packets as its send buffer, to which they need to be copied, is full. None of the transfers can make progress, and thus, a deadlock is reached.

Formally, consider the network as a directed graph where C denotes the set of unidirectional links. Bidirectional links (such as TCP connections) are replaced by 2 opposing unidirectional links. Given a transfer path that utilizes links $e_1, e_2, \cdots, e_n$ ($e_i \in C$) in that order, there is a link dependency between links e_i and e_{i+1} because until the send buffer for e_{i+1} becomes non-full, no packets from e_i can be received. Denote this dependency as $e_i \to e_{i+1}$ ($i = 1, 2, \cdots, n-1$). When there is a cycle in this link dependency graph, a deadlock can potentially occur. This is possible when there are loops in the overlay path, or when multiple indepenent transfers occur concorrently. Thus, in the previous



a. Rank assignment



b. Example path

Figure 4.4: An example of ordered-link routing

example, a deadlock occured due to the cyclic dependencies among links as shown in Figure 4.3(b): $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1$.

4.3.2 Deadlock-free routing basic concepts

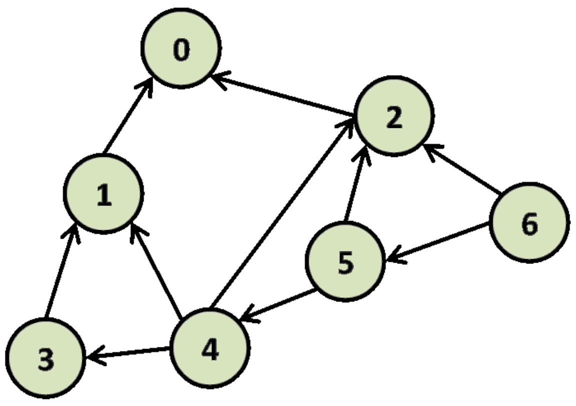
The concept of deadlock-free routing is to compute transfer paths so that there can be no cycles in the link dependency graph. Earlier routing works relied on using regular topologies (such as hypercube or torus) to achieve deadlock-freedom [29, 30]. Here, only algorithms that can be applied to arbitrary irregular topologies are considered.

4.3.3 Ordered-link routing

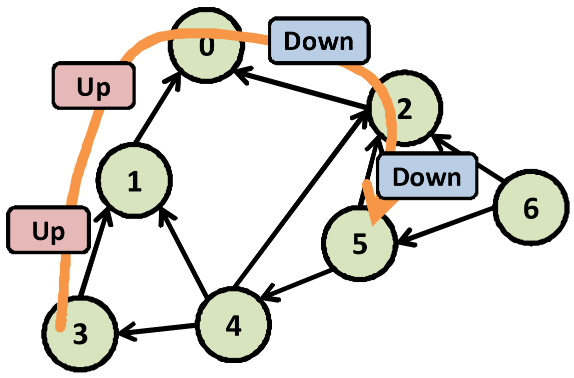
Ordered-link Routing [28] extracts spanning trees with undirected links. Starting from the set of all bidirectional graph links, this method extracts *best-effort* spanning trees until no links are left. It is best-effort because some trees may not be able to span across all nodes. The best-effort spanning-trees are labeled by ranks in ascending order. A rank assignment example is shown in Figure 4.4(a). In order to remove deadlocks, a path, having used any link from a spanning tree of rank r, cannot use any links from spanning trees with rank less than r. Given a network with n nodes and e links, $\lceil \frac{e}{n} \rceil$ ranks will be assigned. Reachability is guaranteed because any source can reach any node only using the spanning tree of the lowest rank. An ordered-link path example is shown in Figure 4.4(b). Because there can be no cyclic dependency between links of differing ranks, and because there can be no cycles within a spanning tree, there can be no deadlock. This method leaves room as to in which order links are assigned to a spanning tree. By default, un-assigned link are chosen at random to form spanning trees in ascending rank.

4.3.4 Up/Down routing

Up/Down routing is the most popular routing scheme currently used in commercial networks [27, 57]. This method assigns a direction to each bidirectonal link in the network and creates a DAG. For a given path, if the path traverses the link along the direction assigned to the link, the path goes



a. Rank assignment



b. Example path

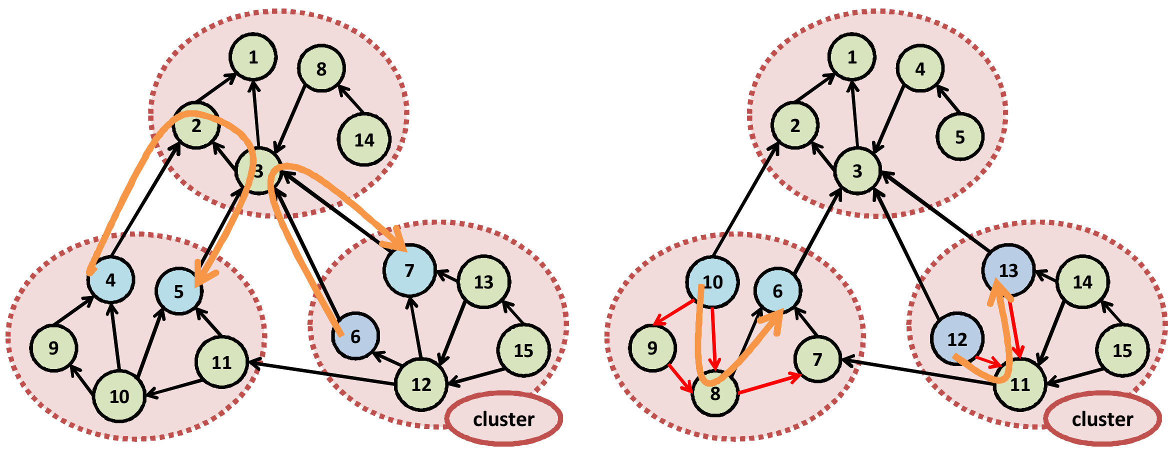
Figure 4.5: An example of Up/Down routing

up (and otherwise *down*). A direction assignment example is shown in Figure 4.5(a). To remove deadlocks, a path cannot go up a link after it has gone down any links. An Up/Down path example is shown in Figure 4.5(b). To ensure reachability, the created DAG must be have a single root sink node. When considering the network to be a directed graph, each undirected link becomes 2 opposing unidirectional-links: one labeled up, the other down. Thus, the algorithm constructs 2 seperate DAGs: one composed of up links, and the other composed of down links. The up DAG is used first, and then the down DAG. No cycling dependencies are possible between up and down unidirectional links, and no dependency cycles exist within each DAG .Thus, there can be no deadlocks. To assign up directions to each link, a root node is assigned, and the network graph is traversed, assigning numerical ids in ascending order. The link direction are assigned towards the node with a smaller node id. The graph traversal is usually done breadth-first [57], but this creates very restrictive constraints, and paths may be prohibited unnecessarily to prevent deadlocks. A depth-first traversal has also been studied to mitigate this effect [56].

4.3.5 Locality-aware Up/Down routing

Traversing the graph breadth-first to assign node ids in Up/Down routing is especially prohibitive for heterogeneous wide-area environments composed of clusters of clusters. In a sparse overlay network, non-adjacent nodes must be reached using 2 or more links. Thus, destinations that are unreachable only using down links must be reached by first going up, and then heading down. This situation is troublesome for nodes that have relatively small ids within a cluster; due to the nature of breadth-first, going up for these nodes implies using inter-cluster wide-area links. But to reach other intra-cluster nodes, it must then come down and re-enter its own cluster. This is illustrated in Figure 4.6(a). The deadlock-free contraints must not restrict route to such paths. Thus, this work proposes the following heuristics to improve path quality for such heterogeneous networks.

- Id assignment graph traversal will be performed depth-first, giving priority to child with greatest proximity



a. Breadth-first assignment   b. Proposed depth-first assignment

Figure 4.6: Up/Down routing in heterogenous networks

The rationale is to remove "sibling" relationships between intra-cluster nodes, where the other node can only be reached by first traversing a common parent node. In an overlay where intra-cluster nodes are densely connected, the likelihood of all intra-cluster nodes having a single anscestory chain is high, and they can be accessed by only traversing up or down links. This removes deadlock-free contraints that force intra-cluster nodes to traverse inter-cluster links. Applying the above heuristic yields much more favorable results for the same example, which is shown in Figure 4.6(b).

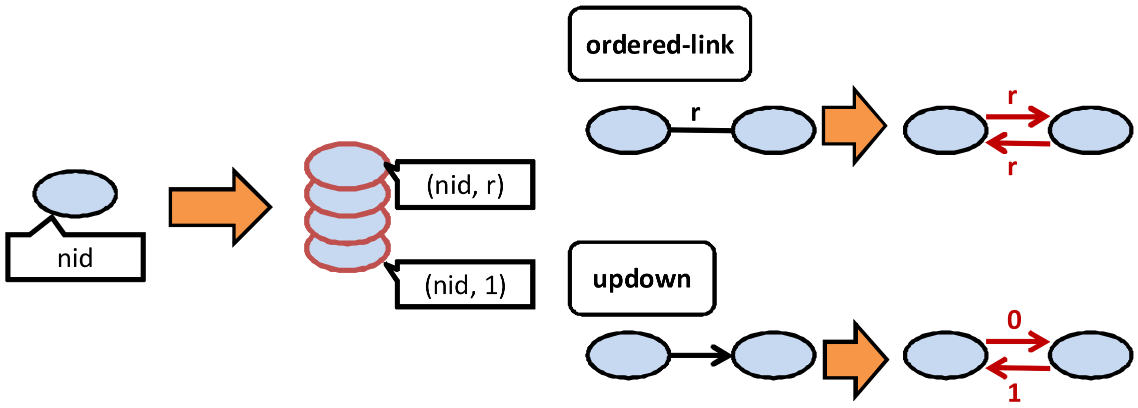


Figure 4.7: Node and edge modification for routing table computation

4.4 Routing Table Construction

The Bellman-Ford algorithm can find optimal routes for Up/Down routing, and approximate ones for ordered-link routing [28]. This work proposes a new deadlock-free routing table computation algorithm that finds optimal routes for both methods in a unified way by extending on Dijkstra's shortest-path routing.

4.4.1 Overview

The problem with directly applying Dijsktra's shortest-path algorithm is the fact that deadlock-free paths depend not only on the current node, but also on history on how the current node was reached. This can be resolved by introducing an integer state i for each node: for Up/Down routing it represents if down links have been previously traversed (0 if no and 1 if yes), and for ordered-link routing it represents the largest link-ranking that have been used so far. Thus, one can envision a routing problem for tuples of node and state value i. Initially, all nodes exchange link states among each other; this can be done scalably using a spanning tree. Next, each node can compute its routing table independently with its node set as the source.

4.4.2 Algorithm

The algorithm takes an arbitrary undirected overlay graph $G = G(N, E)$, where N denotes the set of all nodes and E the set of all bidirectional edges, and modifies it. For all $n \in N$, create r node tuples (n, i) $(i = 1, \cdots, r)$ where $r = 2$ for Up/Down routing and r denotes the *total number of ranks* for ordered-link routing. Denote the set of node tuples as $\tilde{N}$. For all bidirectional edge $e \in E$, replace it with 2 opposing directional edges $\tilde{e}_1$, $\tilde{e}_2$ and label them in the following:

Up/Down: label the edge in the *up* direction 1, and the one in the *down* direction 2

ordered-link: label both $\tilde{e}_1$ and $\tilde{e}_2$ as i where i is the rank of edge e

Denote new edge set as $\tilde{E}$. The node and edge modification is shown in Figure 4.7. As for node

Algorithm 1 *LeveledDijkstra*$(\tilde{N}, \tilde{E}, r, src)$

```
1: // initialize Distance, Predecesor, and source tuples
2: for all $\tilde{n} \in \tilde{N}$ do
3:     $Dist[\tilde{n}] \leftarrow \infty$
4:     $Prev[\tilde{n}] \leftarrow NULL$
5: end for
6: for $i = 1$ to $r$ do
7:     $Dist[(src, i)] \leftarrow 0$
8: end for
9:
10: $Q \leftarrow \tilde{N}$
11: while $|Q| > 0$ do
12:     $(n, i) \leftarrow deleteMin(Dist, Q)$
13:     // iterate over all reachable neighbors of node $n$
14:     for all $\tilde{e} \in outEdges(\tilde{E}, n)$ do
15:         $m \leftarrow \tilde{e}.dst$
16:         $j \leftarrow \tilde{e}.label$
17:         for $k = i$ to $r$ do
18:             // do not consider nodes with
19:             // label smaller than edge label
20:             if $i \leq j \leq k$ then
21:                 $d \leftarrow Dist[(n, i)] + \tilde{e}.dist$
22:                 if $d < Dist[(m, k)]$ then
23:                     $Dist[(m, k)] \leftarrow d$ // decreaseKey(Dist, Q)
24:                     $Prev[(m, k)] \leftarrow (n, i)$
25:                 end if
26:             end if
27:         end for
28:     end for
29: end while
30: return $Prev$
```

connectivity, the following rule is established:

- For all $\tilde{e} : n_s \rightarrow n_d \in \tilde{E}$ with label i, where $n_s, n_d \in N$, $\tilde{e}$ connects all (n_s, j) to (n_d, k) such that $j \leq i \leq k$.

The procedure is shown in Algorithm 1. $\tilde{e}.dst$ and $\tilde{e}.label$ denote the destination node and label for unidirectional edge $\tilde{e} \in \tilde{E}$. As shown in the algorithm, the proposed algorithm only make slight changes to Dijkstra's algorithm. The difference is that the algorithm treats all node tuples $\tilde{n} \in \tilde{N}$ as independent nodes, and that reachable neighbors of a given node is determined based on the connectivity rule explained above. Each node solves it own routing table by setting *src* to itself and initializing distance to (src, i) to 0. The returned *Prev* table can be used to extract paths to each $n \in N$.

4.4.3 Complexity

Using a Fibonacci heap, the complexity of Dijkstra's algorithm is $O(|N|(\log |N| + degree))$, where *degree* denotes the average degree of all nodes. The proposed algorithm obtains a new directed graph

$\tilde{G} = \tilde{G}(\tilde{N}, \tilde{E})$ where $|\tilde{G}| = r|G|$ and $|\tilde{E}| = 2|E|$. Because of the connectivity rule and the replication of nodes, each node's degree increases r folds. Thus, the complexity becomes $O(r|N|\log|N| + r^2|E|)$. For Up/Down routing r is a constant so the complexity becomes $O(|N|\log|N| + |E|)$. For ordered-link routing, $r = \frac{|E|}{|N|}$, thus $O(|E|\log|N| + \frac{|E|^3}{N^2})$. For the worst case, a complete graph, they reduce to $|N|^2$ and $|N|^4$ respectively. Compared with the Bellman-Ford algorithms that are $O(|N||E|)$ per node, this is an improvement for Up/Down routing. Though in the worst case this can be worse than ordered-link routing, this algorithm yields optimal paths rather than approximates [28].

4.5 Utilizing Underlying Network Information

Because the proposed overlay utilize deadlock-free routing algorithms that work on irregular topologies, it functions correctly with arbitrary graphs. But to yield higher performance in LAN/WAN mixed environments, the proposal includes considerations for the underlying network.

4.5.1 Connection selection

The proposed overlay create a locality-aware overlay based on [55] where it utilizes the latency matrix among all nodes. Each node sorts all other nodes in ascending distance. Given n total nodes and a parameter d that determines the connection density, each node attempts to connect to the following nodes:

- all first d closest nodes
- uniform-randomly select d nodes from $[d^k + 1, d^{k+1}]$-th closest nodes for $k = 0, 1, \cdots, \log_d(n)$

Each node densely connect with close nodes, while sparsely with distant nodes. This scheme reduces WAN connections and thus communication contention in WANs for concurrent communication. This is demonstrated in the evaluation.

4.5.2 Routing metric

LAN/WAN mixed environments are complex in the sense that low latency and bandwidth may come in trade-off of each other; maximizing for one may degrade the other. Thus choosing a suitable routing metric requires further investigation. This is beyond the scope of this paper. In the proposed system, sum of the inverse bandwidth of the links is the metric for a path.

Chapter 5

Experiment Results

This section presents evaluation results for the proposed programming framework, and the proposed deadlock-free routing overlay. They were both evaluated in multi-cluster environments connected over a WAN. The detailed descriptions of the environments for the two proposals are given independently.

5.1 Programming Framework Applications on a Large-scale Wide-area Environment

By using the program shown in Figure 3.2(a) as the base, the evaluations to follow show that parallel and distributed applications can be expressed simply in Python using the proposed framework. It shows that the applications can be deployed in a large-scale wide-area environment with minimal effort despite NATs/firewalls, and dynamic fluctuations in compute node resources.

5.1.1 Experiment Condition

First, the details of network environment is outlined. Figure 5.1 and Table 5.1 show the clusters used for the evaluations. It is noteworthy that each cluster has different network administration settings. For example, due to the NAT configuration, most nodes in cluster `kyoto` and `imade` are not accessible from outside. Cluster `kototoi` has global IPs, yet due to the firewall at the gateway router, no incoming connections can be accepted. However, all nodes in all clusters were utilize without any manual configuration; the bootstrapping scheme in Section 3.2.2 automatically bootstrapped all nodes to the peer-to-peer overlay. Exceptions are cluster `istbs` and `tsubame`, in which all its incoming and outgoing packets on most ports are filtered for security reasons. At the time of the experiment, all communication to `istbs` cluster required an SSH-portforwarding via a node inside the cluster. All communication to `tsubame` compute nodes required a 2-hop SSH portforwarding: first via the cluster gateway node, and then to a node inside the cluster. The overlay configuration was setup so as to use SSH-portforwarded TCP connections to these clusters and incorporate them into the overlay network. Once this was realized, communication among all nodes was done without hindrance. The SSH-porforwarding connections to these clusters were initiated from nodes in cluster `hongo`, and all other nodes within `istbs`, and `tsubame` used raw TCP connections to intra-cluster nodes. The gateway node in `tsubame` cluster was further configured to establish SSH-porforwarding connections to intra-cluster

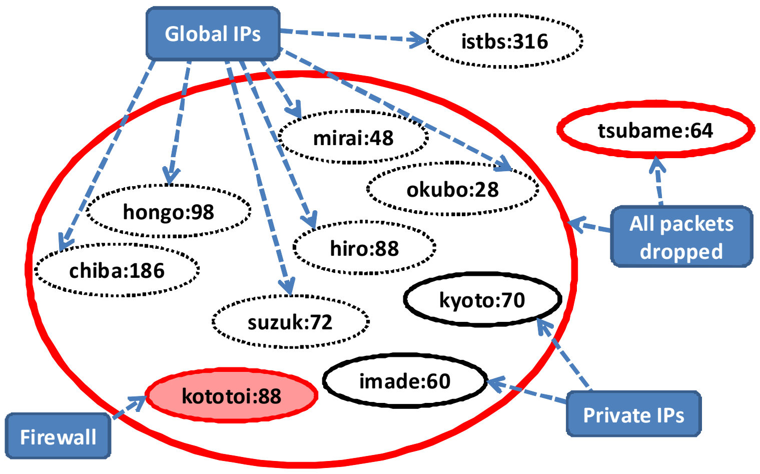


Figure 5.1: Experiment cluster settings denoted by core counts

```
# hongo nodes use SSH to reach istbs nodes
require  hongo  istbs,  prot = ssh, user = <ISTBS-USERNAME>

# hongo nodes use SSH to reach tsubame-gateway node
require  hongo  tsubame-gw, prot = ssh, user = <TSUBAME-USERNAME>

# tsubame-gateway node uses SSH to reach tsubame compute nodes
require  tsubame-gw tsubame-compute, prot = ssh, user = <TSUBAME-USERNAME>
```

Figure 5.2: Connection configuration file used for experiment

compute nodes. The configuration file is shown in Figure 5.2

5.1.2 Fault-Tolerance

The evaluation will show that a simple piece of code as shown in Figure 3.2(a) can carry out a very simple master-worker application despite node insertions and abrupt failures. A single Master object dynamically distributes 10,000 tasks to Worker objects. For node addition, the method described in Section 3.2.2 is used. For node failures, processes on corresponding nodes are simply killed abruptly without warning. The sequence of insertion/deletion is as follows:

0 sec: Initially bootstrap with clusters `hongo`, `chiba`, `suzuk`, `okubo`, `kyoto`, `kototoi` (510 workers)

60 sec: Added cluster `istbs` (203 workers)

280-830 sec: Added cluster `tsubame` (65 workers), but most of them are lost due to slow communication

Table 5.1: Specifications of each cluster

Name	CPU	Nodes (Cores)	Network
chiba	Pentium M 1.86GHz Core2 Duo 2.13GHz	70 (70) 58 (116)	global IP
hongo	Pentium M 1.86GHz Core2 Duo 2.13GHz	70 (70) 14 (28)	global IP
			(firewall)
imade	Core2 Duo 2.13GHz	30 (60)	private IP
okubo	Core2 Duo 2.13GHz	14 (28)	global IP
suzuk	Core2 Duo 2.13GHz	36 (72)	global IP
kyoto	Core2 Duo 2.13GHz	35 (70)	private IP
mirai	Xeon E5410 2.33GHz	6 (48)	global IP
hiro	Xeon E5410 2.33GHz	11 (88)	global IP
istbs	Xeon 2.40GHz	158 (316)	global IP (blocked)
tsubame	Dual Core Opteron 2.40GHz	- (64)	global IP (blocked)

980 sec: Reinserted cluster `tsubame` (65 workers)

1320 sec: All nodes in cluster `chiba` are killed (-169 workers). Some workers whose path between the Master uses cluster `chiba` nodes are also lost.

1800 sec: All nodes in cluster `chiba` are reinserted (+169 workers)

1950 sec: All nodes in cluster `istbs` are killed (-202 workers)

2350 sec: All nodes in cluster `istbs` are reinserted (+202 workers)

No tasks were lost during this experiment. The time series for the number of Workers running, and the number of tasks allocated by the Master to each Worker is shown in Figure 5.3. The figure shows that as the number of workers fluctuates, the master detected task losses and redistributed jobs accordingly. The failure detection latency was in the order of milliseconds. All fault detection is done at the programmer level by detecting faults as exceptions as in Figure 3.2(a). Moreover, it is noteworthy that all participating nodes are interconnected on a TCP overlay and direct connection is not necessary for fault-detection due to the scheme in Section 3.2.3.

5.1.3 Real-life Application

Many real-life parallel and distributed applications require parallel tasks to interact with each other. An example of such applications is a problem solver that uses branch-and-bound. In these applications, it is imperative that all parallel solvers share the latest bound information for efficient computation; these are applications that require periodic communication among nodes. Such applications are impossible to express in programmingless frameworks where inter-task communication is not permitted, or in divide-and-conquer type frameworks where communication is limited to immediate parent and

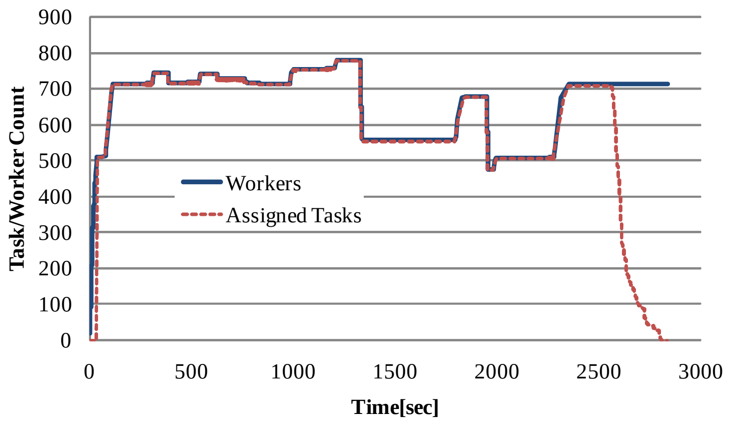


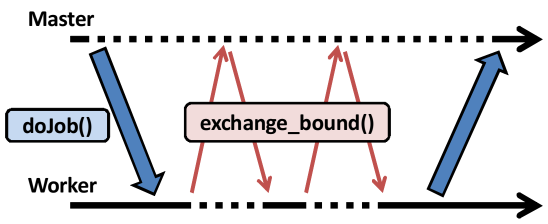
Figure 5.3: Active workers/assigned nodes for FT master-worker

children tasks. As discussed in [15], such applications can be expressed naturally in master-worker models, but there are virtually no frameworks that can handle the hostile network environment (NATs, firewalls, and IP filtering) in the given experimental settings.

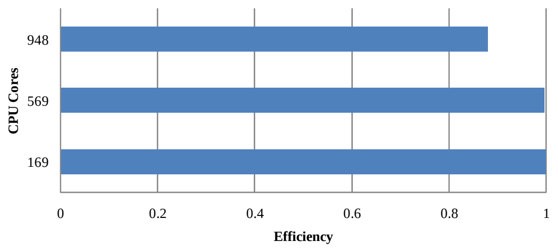
As a case study one of such problems, the Permutation Flowshop Scheduling Problem (P-FSP), was taken and its parallel solver was implemented using the proposed framework. P-FSP is a problem where n jobs have to be processed on m machines in the same order. Only permutation (of jobs) schedules are considered, which means that all jobs have to be processed by machines $0 \dots m-1$ in that order. The goal of this problem is to find a schedule which minimizes the makespan (execution time) with proof. This entails going through the search space of all possible job permutations to find the order with the minimal makespan.

The solver does a parallel branch and bound in a master-worker model, where each worker receives a small section of the search space. The master and the worker programs, which won the 3rd Grid Plugtest Competition [1], had already been implemented in C++. Each node ran a Python program with the framework, and the program communicated with its local Master/Worker C++ program using pipes. The bulk of the computation was delegated to the C++ programs, and the Python program did the master/worker flow control and the necessary communication. Thus, the proposed framework served as the glue to integrate the two and deploy it on the experiment environment.

The flow of the computation is expressed in Figure 5.4(a). When a worker first joins, it first receives a task to solve via an asynchronous RMI. The worker and the master periodically (every 60 seconds) exchange bound information used for the branch-and-bound. The master stores the worker bound information if it is better. Other nodes will update its own latest bound if the query to the master yields better bounds than before. When a worker finishes or aborts its given task, the asynchronous RMI returns and the master gives it its next task. For simple load-balancing, when the master runs out of tasks to distribute, it takes an already assigned task and divides it into smaller tasks to distribute. This is done because the computational size of a task cannot be predetermined; they are simply inferred by how long they have been worked on and become targets for redistribution. The worker aborts its current task if its task has been made into smaller tasks and redistributed to other workers.



a. Master-worker RMI interaction



b. Execution efficiency

Figure 5.4: Evaluation of the Permutation Flowshop Solver

The study shows that the proposed framework allows the programmer to program such applications with minimal expansion to the conventional master-worker application. The only noticeable addition to Figure 3.2(a) is where the worker queries the master periodically using an extra RMI `exchange_bound()`. Because the computation could take months (perhaps years), fault-tolerance was a crucial part of the design. However this is already built into the base sample code in Figure 3.2(a), and failed workers are handled by catching RMI fault exceptions; no additional code was necessary. Moreover, the code required no mutual exclusion and all communication was done in the form of RMIs. The entire code in Python took merely about 250 lines.

Though the modification is simple, it is not necessarily a straightforward modification with other frameworks. Blindly using distributed objects will cause race-conditions at the master node [64], and thus explicit mutual exclusion is required. It will deadlock if the active-object model were used as this is a recursive invocation between the master and the worker [37]. Some master-worker frameworks allow master/workers to send messages to each other [22, 49], and task distribution and bound updates can be done via this way, but this makes the flow and the semantics of the program harder to discern. In the models above, failures cannot be handle at all, or they have to be implemented via seperate failure handlers [22].

The solver was executed on three different configurations:

A: only cluster `chiba`: 168 cores

B: clusters `chiba`, `hongo`, `kototoi`, `okubo`, `suzuk`, `imade`, and `kyoto`: 569 cores

C: clusters `chiba`, `hongo`, `kototoi`, `okubo`, `suzuk`, `imade`, `kyoto`, `istbs`, and `tsubame`: 948 cores

The only necessary network configuration, in the 948 core case, was to specify the SSH portforwarding settings for clusters `istbs` and `tsubame`, which is shown in Figure 5.2. The rest of the deployment was taken care of automatically and successfully created a connected graph of all processes.

An evaluation using a relatively small randomly generated problem instance of ($n = 28$, $m = 20$) is shown here. To measure the performance of the framework, rather than of the algorithm, the computation efficiency is shown. The computation efficiency is calculated as $\frac{C}{NT}$, where N, T, and C resepctively denote the core count, the completion time, and the cummuative computation time across all cores. The results are shown in Figure 5.4(b). With 948 cores across 9 sites, 88% efficiency is maintained.

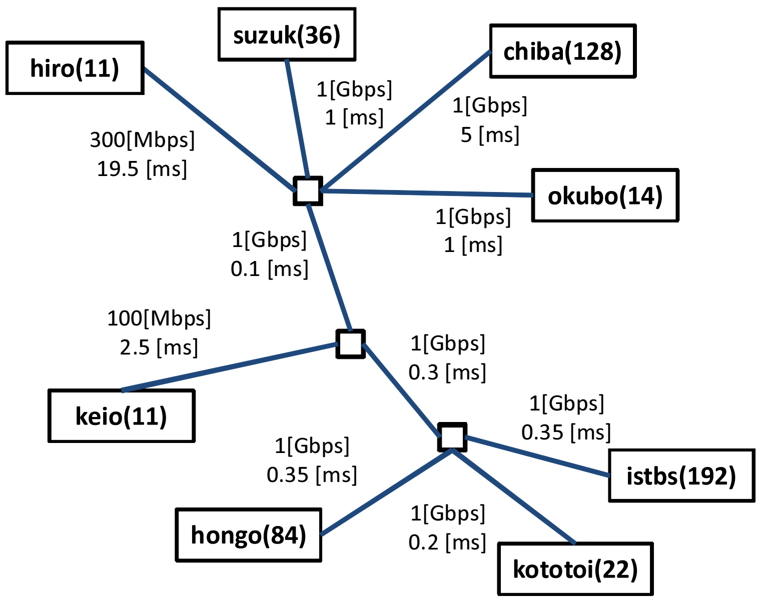


Figure 5.5: Experiment Environment

5.2 Deadlock-free Routing Overlay in Wide-area Networks

This section provides evaluation results for the proposed deadlock-free overlay. In simulation and in large-scale WANs, the evaluation shows that the overhead of using this deadlock-free overlay over deadlock-unaware overlays and direct raw socket communication is very small. The evaluation also shows that the proposed overlay can deliver performance boosts over using direct raw sockets for certain collective communications.

5.2.1 Experiment Condition

Experiments were conducted in both simulation and in a real wide-area environment. The environment consisted of 8 clusters mostly from the InTrigger platform [7]. All nodes use GbE interfaces. The clusters with node counts and its WAN topology information is shown in Figure 5.5. For the evaluation using 10Gbps Myrinet interfaces, the T2K platform [52] was also used. All compute nodes used Linux 2.6.18 and thus, the BIC TCP congestion control algorithm [67].

5.2.2 Deadlock-free Routing Overhead

The evaluation here shows that utilizing the proposed locality-aware Up/Down routing heuristics add very little overhead with respect to deadlock-unaware routing by comparing the following.

path hops: overlay path hop count

bandwidth ratio: bandwidth obtained by using the path in comparison to the direct path

Comparsion for path latency is shown with the point-point experiment to follow. Routing tables for all node pairs were computed by simulation on an overlay of 13 clusters (515 nodes) from InTrigger. The overlay was constructed using the locality-aware scheme with varying connection density. Deadlock-unaware shortest-path routing is also shown as a comparison. Figure 5.6(a) shows that the path lengths between deadlock-unaware and deadlock-free routing is on average minimal. The proposed Up/Down routing heuristics is denoted by *updown-dfs*. Even when the longest path length for all pairs is compared in Figure 5.6(b), though ordered-link routing exhibits very long paths, the proposed Up/Down heuristics yield roughly the same path length as deadlock-unaware routing.

The ratio of bandwidth obtained by using the proposed overlay against that by direct socket communication is also compared. For overlay connection densities including and above 0.03, the average ratio of bandwidth obtained by overlay paths over direct raw sockets exceeded 0.99 for all deadlock-free algorithms. The minimum bandwidth ratios are shown in Figure 5.6(c). Ordered-link routing and Up/Down routing yields some paths with anemic bandwidth for sparse overlays. Analysis showed that these were intra-cluster node pairs who were forced to use narrow inter-cluster links due to the deadlock-free contraints. Because the proposed Up/Down routing heuristics avoid inter-cluster communication for intra-cluter nodes, such poor bandwidth usage is avoided, and is comparable to deadlock-unaware routing.

5.2.3 Point-Point Communication Performance

Evaluation here demonstrates that the proposed deadlock-free overlay performs comparably against direct point-point communication, both for latency and bandwidth utilization. To measure the true round-trip time latency for all paths, the overlay was run on 7 clusters (170 nodes): `hiro`, `istbs`, `okubo`, `suzuk`, `hongo`, `kototoi`, and `chiba`. Using the locality-aware overlay construction, 8% of all possible connections was established. Figure 5.7 shows the CDF of the round trip latency ratio of all node pairs in comparison to when a complete graph was established. As indicated by the horizontal line, 95% of all pairs have latency ratios less than 1.72, 1.51, and 1.48 for ordered-link, updown, and the proposed updown heuristics respectively.

Some overlay paths however traverse inter-cluster links for intra-cluster communication, and experience very large latency stretch factors. This is shown in the distribution of latency on the overlay with respect to that of direct raw sockets in Figure 5.8(a), (b). Such paths arise not because of the deadlock-free constraints, but because of the routing metric selection: *sum of inverse link bandwidths*. Clusters `istbs`, `hongo`, `chiba`, and `suzuk` have 1[Gbps] connections to each other and inter-cluster connections for these yield bandwidth comparable to intra-cluster ones. Thus, these inter-cluster links were selected in exchange for fewer hops. Figure 5.9(a), (b) illustrate that changing the routing metric to the *latency* over the path yields latency comparable to raw socket communication, given the constructed overlay and the deadlock-free routing constraints. It should be noted here that in Figure 5.9(a), Up/Down routing still forces intra-cluster communication to traverse inter-cluster links because of its deadlock-free constraints does not take network locality into account. The proposed Up/Down routing algorithm in Figure 5.9(b) avoids creating such constraints and yields the latency of the underlying network.

Figure 5.10(a),(b),(c) show the bandwidth utilization for long message transfers over varying overlay hops on 3 different environments: (1) within `istbs` cluster (940[Mbps]), (2) across `istbs` and `keio` clusters (95[Mbps]), (3) within T2K (7[Gbps]). The figures in the parenthesis indicate throughput attainable by direct TCP sockets. The results show that comparable bandwidth, even for multiple hops and widely varying interconnects , can be achieved simply by coupling TCP and fixed buffers. Though multiple hops on Myrinet is not up to par, this in reality is not a problem as the locality-aware overlay achieves fewer hops for LAN nodes.

5.2.4 Collective Communication Performance

Not only can using the proposed overlay provide latency and throughput performance equal to direct communication, it can *improve* performance for certain collective communication operations when there is network contention. For all evaluation below, unless otherwise noted, the proposed Up/Down heuritics for routing was used.

5.2.4.1 Gather Communication

Gather is one of many basic collective communications, and using overlays improves gather time, even in high speed networks like a LAN, by removing contention at the switch. In a very simple single cluster of 36 nodes, connected to 1 GbE switch, the collective gather time to a single node for direct communication and the proposed overlay was measured. Evaluation was done with a random graph overlay with varying connection density and with varying message sizes. As in Figure 5.11(a), using the overlay yields higher performance than direct sockets. This is because packets are dropped at the LAN switch as many concurrent packets arrive to be send on the same port. The sender thus has to retransmit the packet after the TCP retransmission timer expires. This is demonstrated by the gather completion time in Figure 5.11(b). The completion time diverge in increments of 200[ms], the default TCP retransmission timer. TCP's fast-retransmission algorithm mitigates this slow down by immediately retranmitting the dropped packet when it receive a duplicate ACK [1] from the receiver. Yet it fails, as in the case here, when the packet lost was the last packet to be sent (no duplicate ACKs to trigger it), or when the fast-retransmitted packet is lost again. When using an overlay to perform gathers, the number of senders to a destination at any given time is restricted, thus reducing packet loss at the switch. To demonstrate this, Figure 5.11(b) shows that sparser overlays yield higher performance.

In multi-cluster environments, contentions at LAN switch by concurrent packet arrivals is mitigated due to a larger distribution in latency, but nonetheless still degrades performance for direct raw sockets. The same operation was done for 291 nodes in 4 clusters: `istbs`, `chiba`, `suzuk`, and `kototoi`, where all data was collected at a node in suzuk cluster. As in Figure 5.11(c), even in this multi-cluster environment, the same phenomenon occurs for short message sizes. The overlay equally or out-performs direct communication constantly.

5.2.4.2 All-All Communication

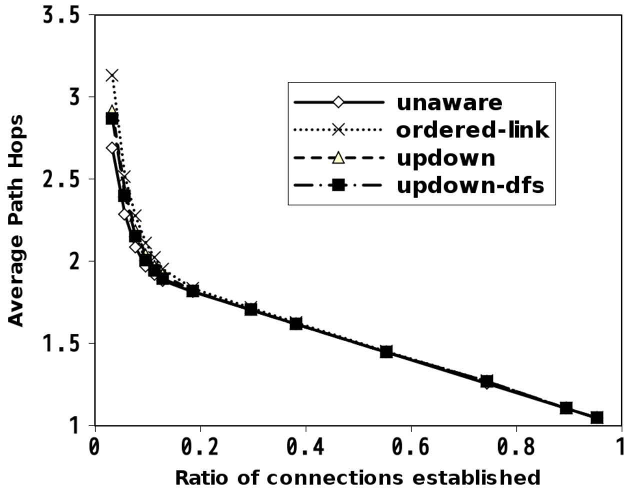
The performance benefit of using overlays over direct sockets are magnified for intensive communication patterns in network environments where bottlenecks exist. This is true for both LAN environments and LAN/WAN mixed environments. The all to all communication time in a single cluster, `istbs` of 177 nodes was measured. This cluster, as shown in Figure 5.12, is connected hierarchically, where there are 14 to 20 nodes per leaf switch. The bottlenecks in this case are the inter-switch Ethernet links shared

[1]duplicate ACKs convey consecutive ACKs with the same packet sequence number to indicate the oldest packet that was lost in transit

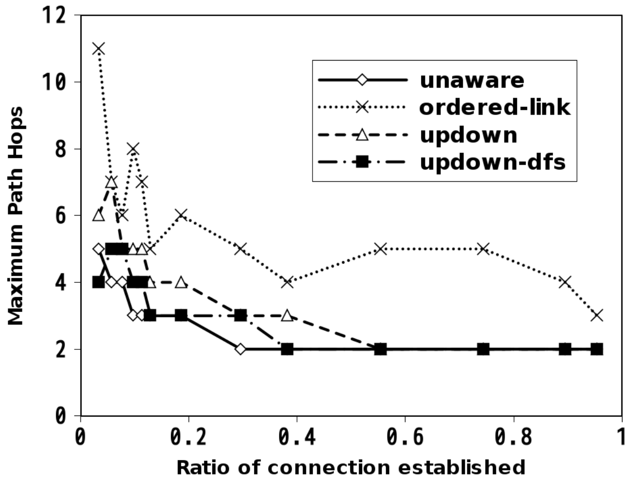
by communication across switches. The overlay was made by the locality-aware scheme, thus fewer connections are made across switches. In this environment, the inter-switch links become the bottleneck and the source of congestion. This is a typical cluster configuration, but Figure 5.13(a) shows that even in such settings, sparse overlays that account for locality can achieve higher performance.

When the WAN-interconnect is used, the performance difference magnifies. The same operation was conducted for 291 nodes in 4 clusters: `istbs`, `chiba`, `suzuk`, and `kototoi`. Results in Figure 5.13(b) arise because much larger number of nodes now contend for much narrower cluster gateway and WAN bottlenecks. This dramatically increases packet loss, and manifests itself in the large performance gap for shorter messages, which are much more vulnerable to packet-loss.

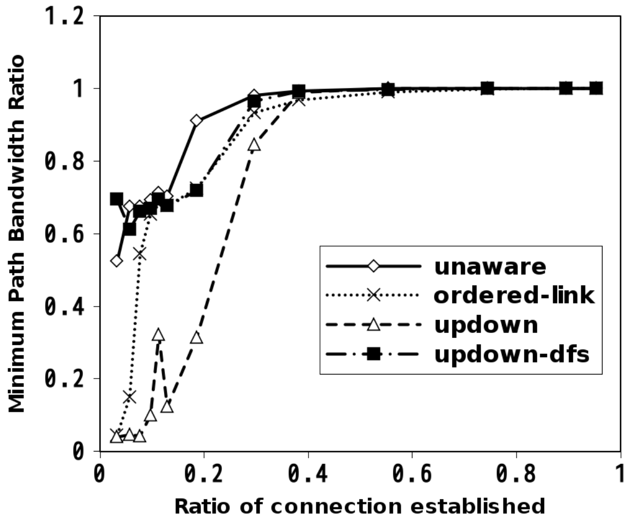
The evaluation here also shows the benefit of using locality-aware overlays as opposed to random graphs. Figure 5.13(c) compares the 10[KB] all to all communication performance difference when the overlay connection selection method is changed to random, instead of locality-aware. The performance diverges as the overlay gets sparser. This arises because the locality-aware scheme eliminates WAN-connections first, while the random graph does not differentiate between LAN/WAN connections. This causes the latter to select a much larger proportion of WAN-connections, and thus giving rise more packet-loss and less performance.



(a) Average Path Hops



(b) Max Path Hops



(c) Min Bandwidth Ratio

Figure 5.6: Simulation results with varying density

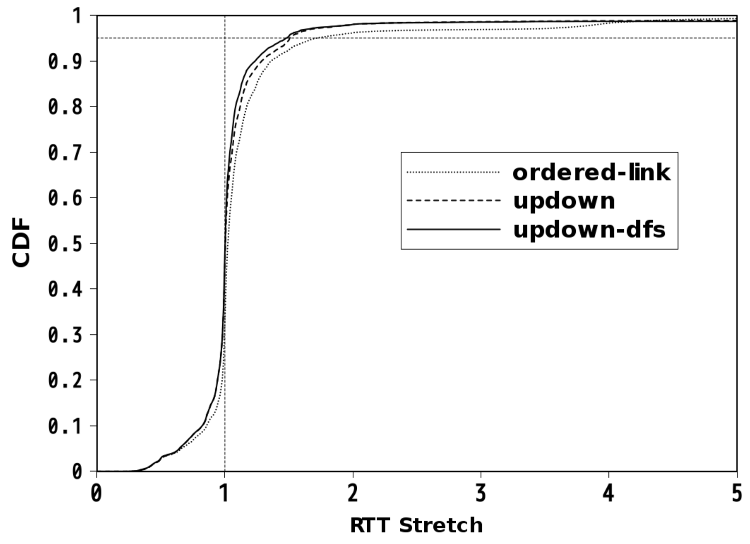
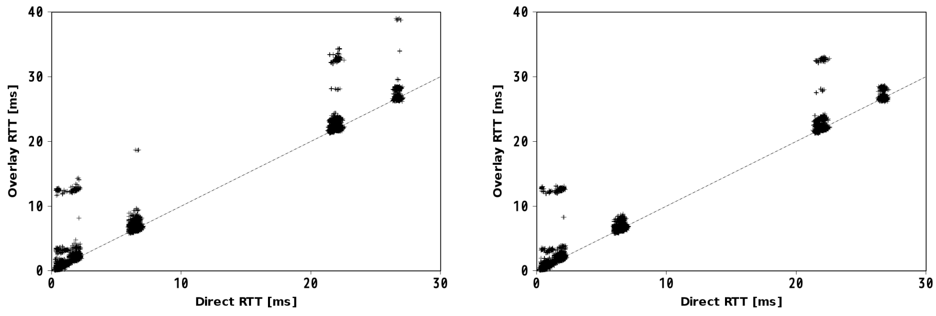
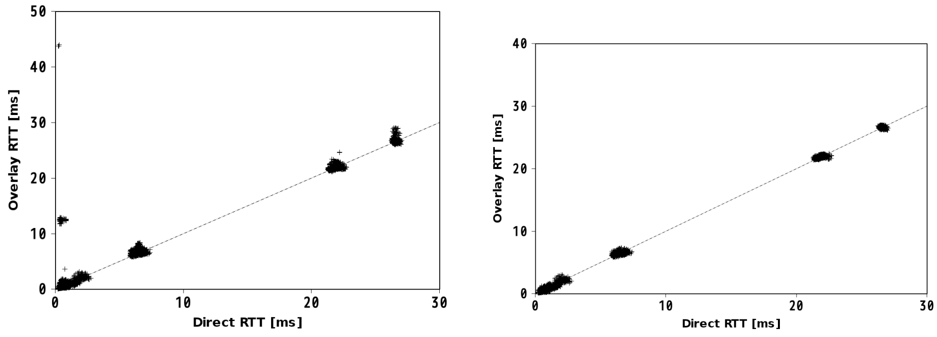


Figure 5.7: Overlay RTT ratio to direct communication



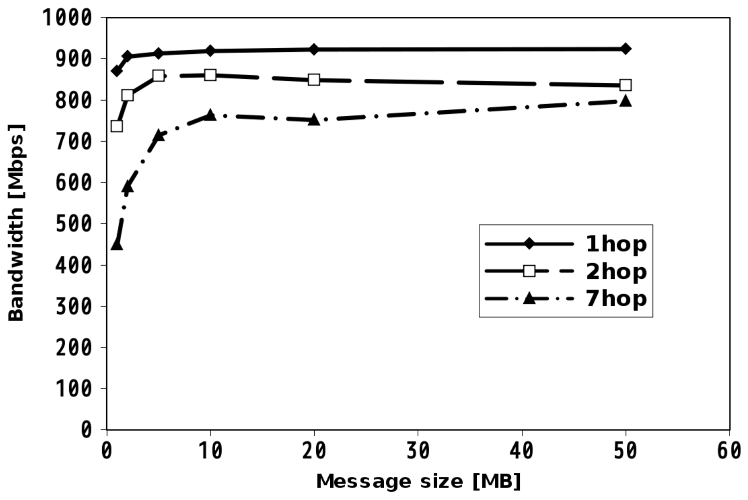
a. Up/Down b. proposed Up/Down

Figure 5.8: RTT ratio with sum of inverse bandwidth metric

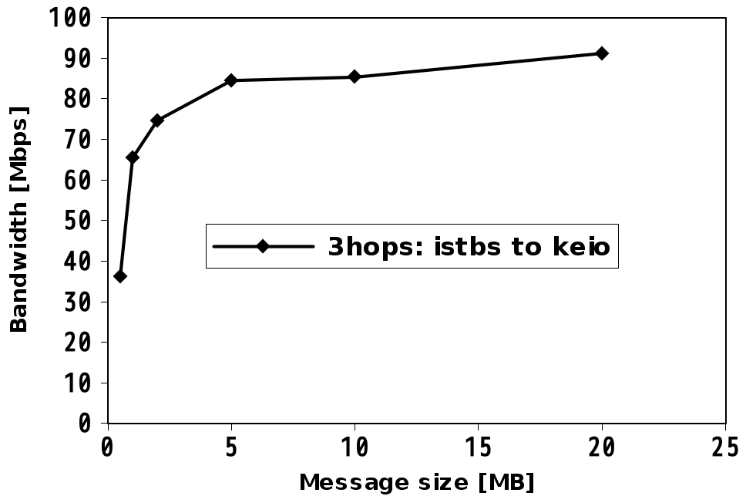


a. Up/Down b. proposed Up/Down

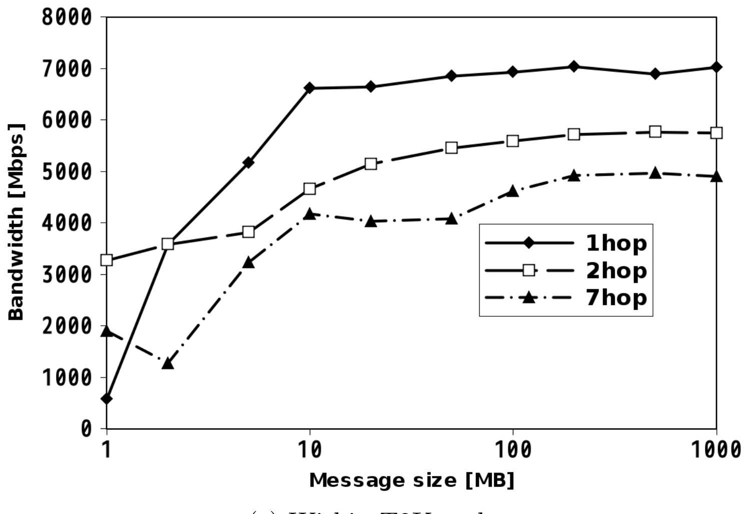
Figure 5.9: RTT distribution with latency metric



(a) Within istbs cluster

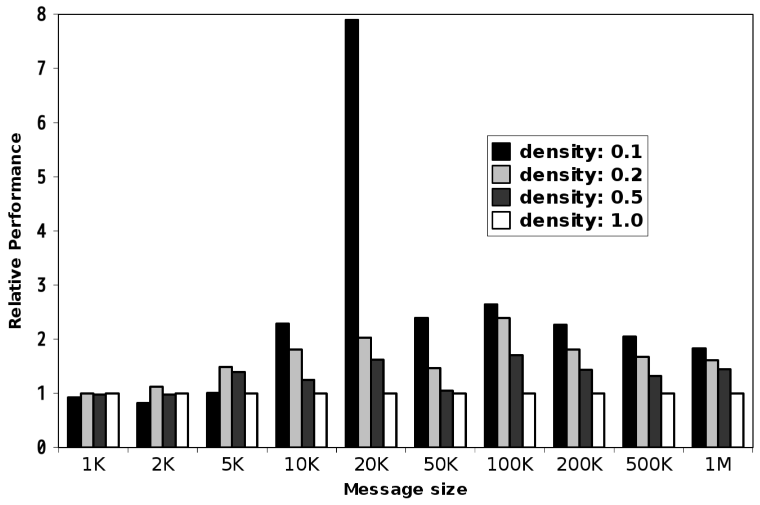


(b) Between kototoi and keio cluster

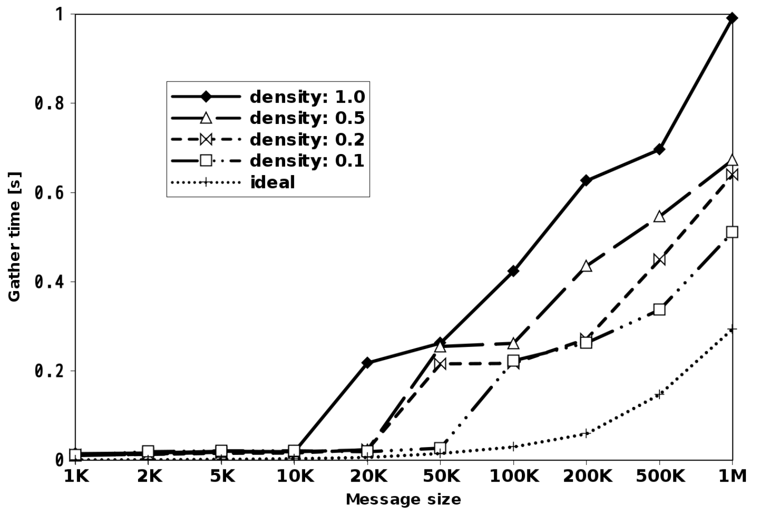


(c) Within T2K nodes

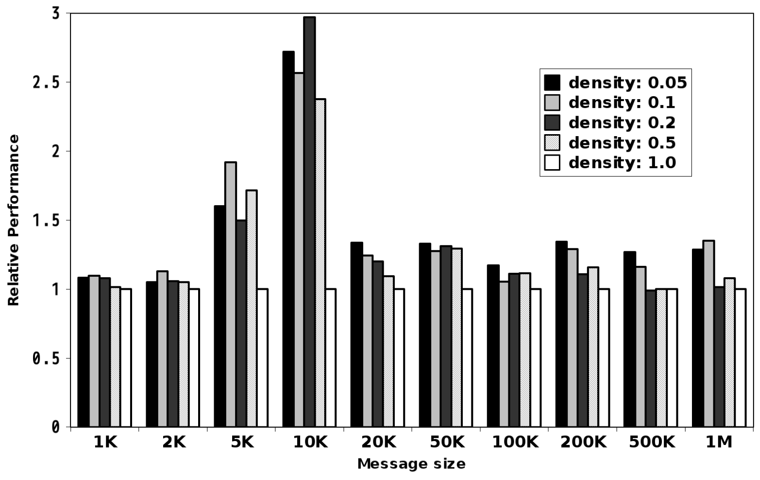
Figure 5.10: Point-Point Long Message Throughput



(a) Performance ratio for 1 cluster



(b) Gather time in 1 cluster



(c) Performance ratio for 4 clusters

Figure 5.11: Gather Performance

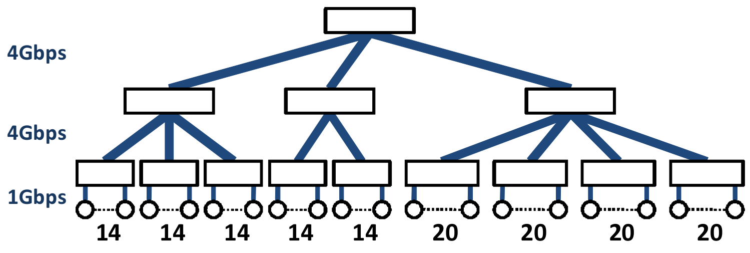
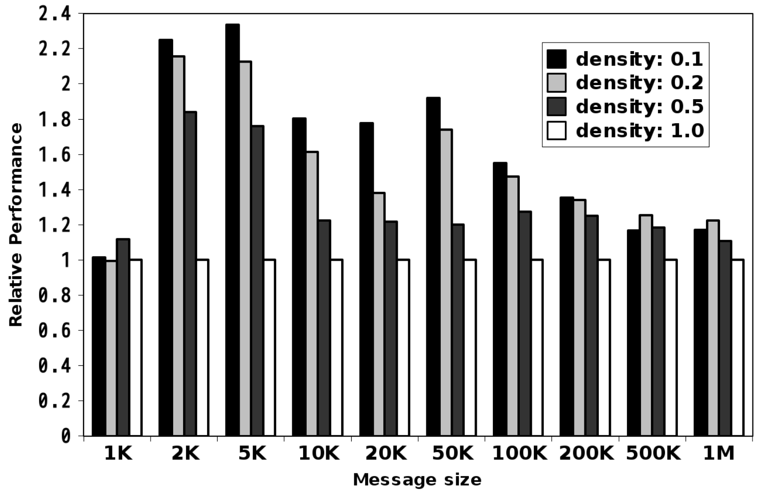
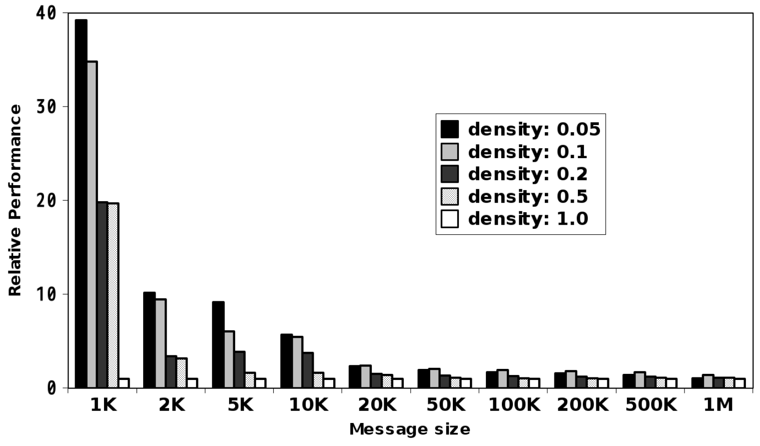


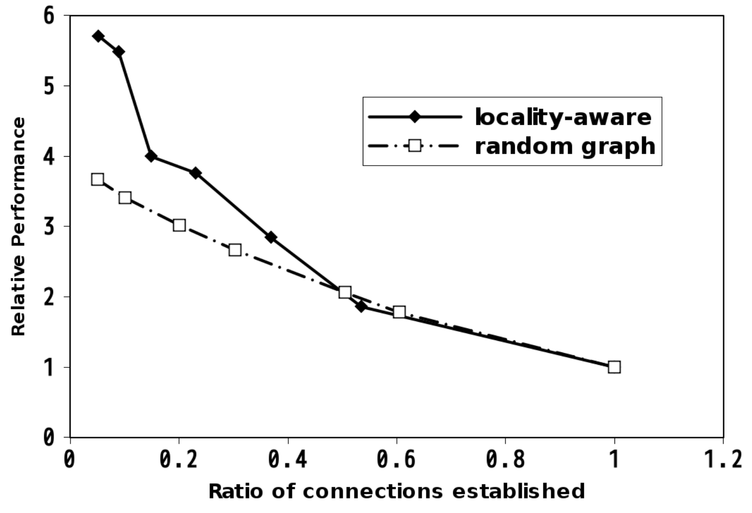
Figure 5.12: Switch topology of cluster used in all to all communication



(a) Performance ratio for 1 cluster



(b) Performance ratio for 4 clusters



(c) Performance comparison by overlay construction scheme

Figure 5.13: All-All Performance

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This work presented 2 contributions to parallel and distributed programming in wide-area environments.

The first proposal presented a programming framework that aims at simple and flexible programming in a large-scale wide-area environment with limited network connectivity, dynamic node joins, and node failures. The framework provides simplicity, without the loss of generality, by extending a widely accepted object-oriented language, Python for wide-area parallel computing. Parallelism is expressed in the form of RMIs with the aid of futures for a natural transition from sequential programs. Using a new invocation serialization semantics, accesses to objects are implicitly serialized without the fear of deadlocks, effectively eliminating locks. The framework introduced simple mechanisms at add nodes to ongoing computation. Node failures are tolerated by abstracting them as RMI exceptions, which can be catch and handled. For connectivity and scalability in WANs, the framework's underlying communication utilizes an automatically constructed TCP overlay, which tolerates dynamic adjustments for node joins and failures. Using the framework, this work demonstrated that:

- the overlay enables computation across 9 clusters with a variety of network environments including NATs and firewalls using minimal user configuration
- the node join/leave features allow whole clusters to be added and removed repeatedly, and the computation will still complete correctly
- taking a branch-and-bound optimization application as an example, the framework enables quick and effective development of parallel applications in large-scale wide-area environments with 900 cores, despite network hindrances like NATs and firewalls

A prototype for gluepy is currently available from its homepage:
http://www.logos.ic.i.u-tokyo.ac.jp/~kenny/gluepy.

The second proposal presented a wide-area, network locality-aware overlay network with a flow control scheme that realizes high transfer throughput for wide-area overlays. The scheme can be implemented straightforwardly by coupling TCP connections with fixed buffer memory and applying deadlock-free routing, thus avoid reimplementing reliability and flow control already provided by TCP.

The simplicity of the scheme allows the overlay to attain straightforwardly the throughput performance of the underlying network. Recognizing the challenges of deadlocks and of using conventional deadlock-free routing to heterogeneous wide-area networks, this work proposed heuristics to adapt deadlock-free routing on the proposed overlay in a heterogeneous wide-area environment. By experiment, this work demonstrated that:

- the proposed deadlock-free routing algorithm imposes a very small latency and throughput performance overhead with respect to deadlock-unaware routing, given suitable information on the underlying network
- the proposed flow control scheme achieves throughput close to that attained by directly using TCP on environments ranging from 100[Mbps] to 10[Gbps] networks
- on certain collective communication, the proposed overlay out-performs standard direct communication by mitigating or completely avoiding network contention

6.2 Future Work

Future work for the proposed high-performance wide-area overlay includes investigating automatic overlay connection management schemes that establish minimal connections to achieve near optimal performance. Such a scheme will no longer require heuristics for locality-aware connection management. Another direction is to implement wide-area applications/libraries using this overlay communication layer, including MPI implementations and distributed file-systems.

The goal of this collective work is to present a programming framework that enables wide-area parallel and distributed computation with node joins/leaves using a simple programming model, combined with a high-performance overlay layer that yields high-performance communication in WANs. Thus, future work for this area is to adapt the routing scheme to accommodate dynamic changes to the overlay. In order to allow overlay node joins and failures during computation, the deadlock-free routing needs to be modified.

References

[1] 3rd Grid Plugtest Report. http://www-sop.inria.fr/oasis/plugtest2006/plugtests_report_2006.pdf.

[2] DAGMan. http://www.cs.wisc.edu/condor/dagman/.

[3] DAS-3. http://www.cs.vu.nl/das3.

[4] distributed.net. http://www.distributed.net.

[5] Grid5000. https://www.grid5000.org.

[6] Infiniband Trade Association. http://www.infinibandta.org.

[7] InTrigger. http://www.intrigger.jp.

[8] OpenPBS. http://www-unix.mcs.anl.gov/openpbs/.

[9] Planetlab. http://www.planet-lab.org.

[10] SINET3. http://www.sinet.jp.

[11] StarPlane. http://www.starplane.org.

[12] Taverna Project Website. http://taverna.sourceforge.net/.

[13] TeraGrid. http://www.teragrid.org.

[14] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[15] K. Aida and T. Osumi. A Case Study in Running a Parallel Branch and Bound Application on the Grid. In *SAINT '05: Proceedings of the 2005 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 164–173, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Y. Amir, B. Awerbuch, C. Danilov, and J. Stanton. Global Flow Control for Wide Area Overlay Networks: a Cost-Benefit Approach. In *OPENARCH 2002: Proceedings of the 5th IEEE Conference on Open Architectures and Network Programming*, pages 155–166, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Y. Amir and C. Danilov. Spines. http://www.spines.org.

[18] Y. Amir and C. Danilov. Reliable Communication in Overlay Networks. In *DSN 2003: Proceedings of the 33rd Annual IEEE International Conference on Dependable Systems and Networks*, pages 511–520, Washington, DC, USA, June 2003. IEEE Computer Society.

[19] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 131–145, New York, NY, USA, 2001. ACM.

[20] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[21] J. K. Antonio. Concurrent Communication in High-Speed Wide Area Networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):264–273, 1994.

[22] H. Aoki, H. Nakada, K. Tanaka, and S. Matsuoka. A Programming Environment with Dynamic Node Configuration for Hierarchical Grid: Jojo2 (in Japanese). In *SACSIS '06: Proceedings of the Annual IPSJ Symposium on Advanced Computing Systems and Infrastructures*, pages 101–108, Tokyo, Tokyo, Japan, 2006. Information Processing Society of Japan.

[23] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput-Competitive On-Line Routing. In *FOCS 1993: Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 32–40, Washington, DC, USA, November 1993. IEEE Computer Society.

[24] S. Ayyub, D. Abramson, C. Enticott, S. Garic, and J. Tan. Executing Large Parameter Sweep Applications on a Multi-VO Testbed. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 73–82, Washington, DC, USA, 2007. IEEE Computer Society.

[25] P. E. S. Barbosa, C. L. Rodrigues, J. C. A. Figueiredo, and D. D. S. Guerrero. Distributed Verification of Occurrence Graphs: Investigating the Use of Computational Grids. In *IECON 2007: Proceedings of the 33rd Annual Conference of the IEEE Industrial Electronics Society*, pages 82–87, Washington, DC, USA, November 2007. IEEE Computer Society.

[26] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[27] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. king Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15:29–36, 1995.

[28] D. M. Chiu, M. Kadansky, R. Perlman, J. Reynders, G. Steele, and M. Yuksel. Deadlock-Free Routing Based on Ordered Links. In *LCN '02: Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, pages 62–71, Washington, DC, USA, 2002. IEEE Computer Society.

[29] W. J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):466–475, 1993.

[30] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, 36(5):547–553, 1987.

[31] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[32] A. Denis, C. Pérez, and T. Priol. PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes. *Future Generation Computer Systems*, 19(4):575–585, 2003.

[33] M. T. Egner, M. Lorch, and E. Biddle. UIMA Grid: Distributed Large-scale Text Analysis. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.

[34] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[35] T. Fuhner, S. Popp, and T. Jung. A novel framework for distributing computations. DisPyTE – distributing Python tasks environment. *Journal of Computational Electronics*, 5(4):349–352, December 2006.

[36] Y. Horita, K. Taura, and T. Chikayama. A Scalable and Efficient Self-Organizing Failure Detector for Grid Applications. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 202–210, Washington, DC, USA, 2005. IEEE Computer Society.

[37] F. Huet, D. Caromel, and H. E. Bal. A High Performance Java Middleware with a Real Application. In *SC 2004: Proceedings of the 2004 ACM/IEEE Supercomputing Conference*, pages 2–17, Washington, DC, USA, November 2004.

[38] K. Kar, S. Sarkar, and L. Tassiulas. A Simple Rate Control Algorithm for Maximizing Total User Utility. *INFOCOM 2001: Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies*, 1:133–141, 2001.

[39] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *PPoPP '99: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 131–140, New York, NY, USA, 1999. ACM.

[40] M. Koibuchi, A. Funahashi, A. Jouraku, and H. Amano. L-Turn Routing: An Adaptive Routing in Irregular Networks. In *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*, pages 383–392, Washington, DC, USA, 2001. IEEE Computer Society.

[41] G. Kola and M. Livny. DiskRouter: A Flexible Infrastructure for High Performance Large Scale Data Transfers. Technical Report Technical Report CS-TR-2003-1484, University of Wisconsin, 2003.

[42] M. Kuhn, S. Schmid, and R. Wattenhofer. Distributed Asymmetric Verification in Computational Grids. In *IPDPS 2008: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, Washington, DC, USA, April 2008. IEEE Computer Society.

[43] S. Kunniyur and R. Srikant. End-to-end Congestion Control Schemes: Utility Functions, Random Losses and ECN Marks. *IEEE/ACM Transactions on Networking*, 11(5):689–702, 2003.

[44] G.-I. Kwon and J. W. Byers. ROMA: Reliable Overlay Multicast with Loosely Coupled TCP Connections. In *INFOCOM 2004: Proceedings of the 23rd Conference of the IEEE Communications Society*, pages 385–395, March 2004.

[45] S.-J. Lee, S. Banerjee, P. Sharma, P. Yalagandula, and S. Basu. Bandwidth-Aware Routing in Overlay Networks. In *INFOCOM 2008. Proceedings of the 27th Conference on Computer Communications*, pages 1732–1740, April 2008.

[46] J. Linderoth, J.-P. Goux, and M. Yoder. Metacomputing and the Master-Worker Paradigm. Technical Report ANL/MCS-P792-0200, Mathematics and Computer Science Division, Argonne National Laboratory, February 2000.

[47] J. Maassen and H. E. Bal. Smartsockets: solving the connectivity problems in grid computing. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2007. ACM.

[48] S. Naganuma, K. Takahashi, H. Saito, T. Shibata, K. Taura, and T. Chikayama. Improving Efficiency of Network Bandwdith Estimation Using Topology Information (in Japanese). In *SACSIS '08: Proceedings of the Annual IPSJ Synposium on Advanced Computing Systems and Infrastructures*, pages 359–366, Tokyo, Tokyo, Japan, 2008. Information Processing Society of Japan.

[49] H. Nakada and S. Matsuoka. A Java-based Programming Environment for hierarchical Grid: Jojo. In *CCGRID '04: Proceedings of the Fourth IEEE International Symposium on Cluster Computing and the Grid*, pages 51–58, 2004.

[50] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *SIGCOMM '03: Proceedings of the 2003 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 11–18, New York, NY, USA, 2003. ACM.

[51] A. Nakao, L. Peterson, and A. Bavier. Scalable routing overlay networks. *SIGOPS: Operating Systems Review*, 40(1):49–61, 2006.

[52] H. Nakashima. T2K Open Supercomputer: Inter-University and Inter-Disciplinary Collaboration on the New Generation Supercomputer. In *ICKS '08: Proceedings of the International Conference on Informatics Education and Research for Knowledge-Circulating Society*, pages 137–142, Washington, DC, USA, 2008. IEEE Computer Society.

[53] J. X. Parreira, C. Castillo, D. Donato, S. Michel, and G. Weikum. The Juxtaposed approximate PageRank method for robust PageRank approximation in a peer-to-peer web search network. *The VLDB Journal*, 17(2):291–313, 2008.

[54] C. E. Perkins and E. M. Royer. Ad-hoc On-Demand Distance Vector Routing. pages 90–100, February 1999.

[55] H. Saito and K. Taura. Locality-aware Connection Management and Rank Assignment for Wide-area MPI. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 249–258, Washington, DC, USA, 2007. IEEE Computer Society.

[56] J. C. Sancho, A. Robles, and J. Duato. An Effective Methodology to Improve the Performance of the Up*/Down* Routing Algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):740–754, 2004.

[57] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: a High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications*, 9:1318 – 1335, 1991.

[58] T. Shirai, H. Saito, and K. Taura. A Fast Topology Inference: A building block for network-aware parallel processing. In *HPDC '07: Proceedings of the 16th International Symposium on High Performance Distributed Computing*, pages 11–22, New York, NY, USA, 2007. ACM.

[59] K. Shudo, Y. Tanaka, and S. Sekiguchi. P3: P2P-based Middleware Enabling Transfer and Aggregation of Computational Resources. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid*, pages 259–266, Washington, DC, USA, 2005. IEEE Computer Society.

[60] K. Taura. GXP : An Interactive Shell for the Grid Environment. In *IWIA 2004: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 59–67, Washington, DC, USA, 2004. IEEE Computer Society.

[61] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f : A Future-Based Polymorphic Typed Concurrent Object-Oriented Language – Its Design and Implementation. In *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, pages 275–292. American Mathematical Society, 1994.

[62] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[63] G. Urvoy Keller and E. W. Biersack. A Congestion Control Model for Multicast Overlay Networks and its Performance. In *NGC 2002: Proceedings of the 4th International Workshop on Network Group Communication*, October 2002.

[64] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, 2005.

[65] K. Verstoep, J. Maassen, H. E. Bal, and J. W. Romein. Experiences with Fine-Grained Distributed Supercomputing on a 10G Testbed. In *CCGRID '08: Proceedings of the Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 376–383, Washington, DC, USA, 2008. IEEE Computer Society.

[66] G. Wrzesinska, R. V. van Nieuwpoort, J. Maassen, T. Kielmann, and H. E. Bal. Fault-tolerant Scheduling of Fine-grained Tasks in Grid Environments. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(1), 2006.

[67] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. *INFOCOM 2004: Proceedings of the 23rd Conference of the IEEE Communications Society*, 4:2514–2524, March 2004.

[68] A. Young, J. Chen, Z. Ma, A. Krishnamurthy, L. Peterson, and R. Y. Wang. Overlay Mesh Construction Using Interleaved Spanning Trees. In *INFOCOM 2004: Proceedings of the 23th Conference of the IEEE Communications Society*, volume 1, March 2004.

Publications

Journal Articles

1. Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura. A Framework for Flexible Programming in Complex Grid Environments. *IPSJ Transactions on Advanced Computing Systems*, Vol.1 No.2 (ACS 23), pp.157-168, August 2008 (in Japanese).

2. Ken Hironaka, Kenjiro Taura, and Takashi Chikayama. A Low-stretch Object Migration Scheme for Wide-are Environments. *IPSJ Transactions on Programming*, Vol.48 No.SIG 12 (PRO 34), pp.28-40, August 2007.

Refereed Papers

1. Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura. gluepy: A Simple Distributed Python Framework for Complex Grid Environments. In *21st Annual International Workshop on Languages and Compilers for Parallel Computing (LCPC 2008)*, LNCS Vol.5335, pp.249-263, Edmonton, Canada, July 2008.

2. Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura. A Framework for Flexible Programming in Complex Grid Environments. In *Proceedings of the Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2008)*, pp.349-358, Tsukuba, Japan, May 2008 (in Japanese).

Unrefereed Papers

1. Ken Hironaka, Hideo Saito, and Kenjiro Taura. Deadlock-free Routing in Wide-area TCP Overlays. *IPSJ SIG Technical Report OS-109 (SWoPP 2008)*, pp.9-15, Saga, Japan, August 2008 (in Japanese).

2. Yasuhito Takamiya, Ken Hironaka, Hideo Saito, and Kenjiro Taura. An Efficient Management Method for Multi-site Distributed Cluster Environments. *IPSJ SIG Technical Report OS-108*, pp.131-138, Naha, Japan, April 2008 (in Japanese).

3. Ken Hironaka, Shogo Sawai, and Kenjiro Taura. A Distributed Object-Oriented Library for Computation Across Volatile Resources. *IPSJ SIG Technical Report OS-106 (SWoPP 2007)*, pp.71-78, Asahikawa, Japan, August 2007 (in Japanese).

4. Hideo Saito, Yoshikazu Kamoshida, Shogo Sawai, Ken Hironaka, Kei Takahashi, Sekiya Takeshi, Nan Dun, Takeshi Shibata, Daisaku Yokoyama, and Kenjiro Taura. InTrigger: A Multi-site Distributed Computing Environment Supporting Flexible Configuration Changes. *IPSJ SIG Technical Report HPC-111 (SWoPP 2007)*, pp.237-242, Asahikawa, Japan, August 2007 (in Japanese).

Poster Presentations

1. Ken Hironaka, Hideo Saito, Kei Takahashi, and Kenjiro Taura. A Framework for Flexible Programming in Complex Grid Environments. At *8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008).*