UNIVERSITY OF CALIFORNIA, SAN DIEGO

IMPROVING SCALABILITY AND FAULT TOLERANCE IN AN APPLICATION MANAGEMENT INFRASTRUCTURE

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science
in
Computer Science

by

Nickolay Topilski

Committee in charge:

Professor Amin Vahdat, Chair
Professor Alex C. Snoeren
Professor George Varghese
Professor Jeannie Albrecht

2008

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to acknowledge Professor Amin M. Vahdat for his support and guidance which proved to be invaluable in conducting this work.

I would also like to acknowledge Dr. Jeannie Albrecht who through support, motivation, and input made this work possible.

I would also like to acknowledge Charles E. Killian, Jr., whose advice and input on Mace was essential in accomplishing the task at hand.

This Thesis, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

ABSTRACT OF THE THESIS

Improving Scalability and Fault Tolerance in an Application Management Infrastructure

by

Nickolay Topilski
Master of Science in Computer Science
University of California, San Diego, 2008
Professor Amin Vahdat, Chair

This Thesis explores the challenges associated with distributed application management in large-scale computing environments. In particular, we investigate several techniques for extending Plush, an existing distributed application management framework, to provide improved scalability and fault tolerance without sacrificing performance. One of the main limitations of Plush is the structure of the underlying communication fabric. We explain how we incorporated the use of an overlay tree provided by Mace, a toolkit that simplifies the implementation of overlay networks, in place of the existing communication subsystem in Plush to improve robustness and scalability.

Chapter 1

Introduction

Scalability has always been an important aspect of distributed system design. However, as the cost of computers continues to drop and the availability of large-scale computing platforms like PlanetLab [21, 6], Amazon EC2 [11], and the Teragrid [9] continues to rise, system designers must revisit what it means to make an application scale. Today it is common for large-scale clusters to contain thousands of computers or more, providing petabytes of data storage and hundreds of teraflops of computing capabilities. While clusters of this size and capacity have the potential to significantly improve an application's performance, designing and deploying applications that can scale to thousands of computers and fully utilize all available resources is difficult.

Aside from the difficulties associated with designing applications that can achieve the maximum performance possible in large-scale platforms, another often overlooked problem is configuring and managing the computers that will host the application. Even if the system designer successfully builds a scalable application, deploying, monitoring, and debugging the application running on thousands of high-performance computers introduces many new challenges. This is especially true in computing platforms where there is no common file system running on the computers. In these environments the developer must manually copy and install the necessary software for running the application to the cluster resources and cope with any problems or failures that occur during the file transfer and installation processes. Only after all software has been successfully installed and the computers correctly configured can the execution begin.

After the execution begins, developers need a way to detect and, if possible, recover from failures to ensure that the application runs to completion. When running an application on thousands of resources[1], failures—including both application-level failures and host- or network-level failures—are inevitable. If a failure is detected, the steps required for recovery vary, and largely depend on application-specific semantics. For some applications, recovery may simply be a matter of restarting a failed process on a single computer. For other applications, a single failure may require the entire application running across thousands of cluster resources to be aborted and restarted. For all applications, one fact holds true: a quick and efficient method for failure detection and recovery is needed for successful execution.

Distributed application management systems help ease the burdens associated with deploying and maintaining distributed applications in large-scale computing environments. They are designed to simplify many tasks associated with managing large-scale computations, including software installation, failure detection and recovery, and execution management. The goal is to hide the underlying details related to resource management and create a user-friendly way to manage distributed applications running on thousands of computers. In short, application management infrastructures allow the system designer to focus on fine-tuning the performance of their application, rather than managing and configuring the resources on which the application will run. There is, however, an important caveat regarding the use of an application management system: the management system itself must reliably scale at least as well as the application being managed.

Many existing application management systems scale to one or two hundred computers, but few achieve the scalability required in today's large-scale computing clusters. In addition, several of the existing systems focus on tasks associated with system administration, often making assumptions about the homogeneity of the computing resources that are not always valid in clusters with thousands of computers. Motivated

[1]Throughout this Thesis, the term *resource* is used to generally describe any device capable of hosting an application. The most commonly used resource is simply a computer.

by the limitations of existing approaches, our intent is to design and implement a scalable application management infrastructure that accomplishes the following four goals:

1. **User-friendliness** - The infrastructure must be easy to use and easily adapted for a variety of applications.

2. **Heterogeneous resource support** - The system must be capable of execution in heterogeneous computing environments.

3. **Fault tolerance** - The management system must automatically detect and recover from application and resource failures.

4. **Scalability** - The infrastructure must scale to hundreds or thousands of resources.

Rather than start from scratch, we chose to use an existing application management system as a basis for our work. In particular, we chose Plush [2, 4], a distributed application management framework initially designed for PlanetLab. Plush provides several user-friendly interfaces and supports execution in a variety of environments, thus accomplishing the first two of our four goals. Plush does not provide adequate scalability or fault tolerance, however, based on the size of large-scale computing platforms in use today. The main problem in Plush is the design of the underlying communication infrastructure. The simple design achieves good performance and fault tolerance in small clusters, but is not resilient to failures and does not scale beyond approximately 300 resources. Hence, the majority of our work centers on addressing these limitations of Plush.

To this end, this Thesis describes the extensions required to improve the scalability and fault tolerance of Plush, and introduces an enhanced system called Plush-M. Plush-M extends Plush to incorporate a scalable and robust overlay tree in place of the existing star-based communication subsystem. We leverage the capabilities of Mace [17], a C++ language extension and library for building distributed systems, to build our overlay tree. Since Mace provides many useful features for designing scalable and robust overlay networks, the integration of Plush and Mace enables us to experiment with many different types of overlays for the communication infrastructure in Plush to ultimately determine the best design.

The remainder of this Thesis is organized as follows. Chapter 2 describes an overview of Plush and Mace, and motivates our design of Plush-M. Chapter 3 discusses some of the problems that had to be addressed during the implementation of Plush-M, and how we overcame these issues. Chapter 4 provides a preliminary evaluation of the scalability and fault tolerance of Plush-M, while Chapter 5 discusses related work. Finally, Chapter 6 describes future directions that we plan to explore and Chapter 7 concludes.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Chapter 2

Plush and Mace: A Brief Overview

This chapter briefly describes Plush and Mace. Since the design and implementation of Plush-M largely consists of the integration of Plush and Mace, it is helpful to understand what Plush and Mace do in isolation before discussing the design of Plush-M in Chapter 3.

2.1 Plush

Plush is a generic, distributed application management infrastructure that provides a set of “building-block” abstractions for specifying, deploying, and monitoring distributed applications. The building-block abstractions are part of an extensible application specification language that Plush uses to define customized flows of control for distributed application management. By defining a custom set of blocks for each application, developers can specify exactly how their application should be executed and monitored, including important behaviors related to failure recovery for application-level errors detected by the Plush monitoring service. Plush also provides advanced support for multi-phased applications (which are common in grid environments) through a set of relaxed synchronization primitives in the form of partial barriers [3]. Partial barriers help applications cope with “straggler” computers by automatically detecting bottlenecks and remapping uncompleted computations as needed. Applications that use partial barriers achieve improved performance in failure-prone environments, such as wide-area testbeds.

Plush's architecture primarily consists of two key components: the controller and the clients. The Plush controller, which usually runs directly on the application developer's desktop computer (also called the control node), issues instructions to the clients to help direct the flow of control for the duration of the execution. All computers aside from the controller that are involved in the application become Plush clients (or participants). At the start of an execution, all clients connect directly to the server using standard TCP connections. Plush maintains these connections for the lifetime of the application. The controller then directs the execution of the application by issuing commands to each client which are then executed on behalf of the developer. This simple client-server design results in quick and efficient failure detection and recovery, which are both important aspects of application execution management.

The original design of Plush targeted applications running on PlanetLab. PlanetLab is a volatile, resource-constrained wide-area testbed with no distributed file system. The initial version of Plush provided a minimal set of automated failure recovery mechanisms to help application developers deal with the most common problems experienced on PlanetLab. Over the past few years, Plush was extended to provide generic application management support in a variety of computing environments. In particular, Plush now supports execution on Xen virtual machines [5], emulated ModelNet virtual clients [22], and cluster computers. However, since the size of PlanetLab during the initial development of Plush was approximately 400 computers worldwide, the current design does not scale well beyond a few hundred computers. Thus before Plush will be useful in large-scale clusters, scalability must be revisited.

2.1.1 Plush Communication Subsystem

While a detailed discussion of how Plush manages distributed applications is outside of the scope this Thesis, it is important to understand why Plush does not provide the scalability or fault tolerance required in large-scale computing platforms. The main scalability bottleneck in Plush is the communication subsystem. By default, all clients participating in the execution of an application connect directly to the controller, forming a star topology. This star topology, in the majority of the cases, provides optimal performance, since all of the clients directly link to the control node. Furthermore, the star topology is easy to maintain, because fault handling in the communication fabric is relatively simple. For example, in the case of a single client failure, only the failed client is affected, and the failure is readily detected by the central controller using the direct connection. Unfortunately, there is a trade-off between performance and scalability. The star topology scales well to approximately 300 nodes, but the topology ceases to function on larger topologies once the OS-defined file descriptor limit per process is reached on the control node. In addition, resources like spare CPU cycles, bandwidth, and latency required to communicate with client machines become a bottleneck at the control node.

For applications running in large-scale clusters, Plush was extended to use a simple tree topology for communication rather than a star. The tree topology supports a maximum depth of only two levels, which results in "bushy" trees where each non-leaf node has a large number of children. The goal was to reduce the number of connections to the control node while also minimizing the number of network hops between the controller and all clients. Unfortunately, while the basic tree topology achieves relatively good performance and scalability, it is not very robust to failures. The failure of intermediate and leaf nodes can lead to unbalanced trees, unaccounted loss of participant nodes, and large tree reconfiguration penalties. Furthermore, imposing a maximum tree depth limitation of two requires intermediate nodes (children of the root) to have sufficient CPU and bandwidth resources available to support large numbers of children (leaf nodes), similar to the resource requirements necessary at the control node.

Thus, in failure prone environments Plush is essentially limited to utilizing the star topology with 300 nodes or less. The only way to use the tree topology effectively is to carefully select reliable and well-connected non-leaf nodes for successful execution of distributed applications. In order to use Plush in large-scale environments, the shortcomings of this tree building mechanism must be addressed.

2.2 Mace

Mace is a C++ language extension and source-to-source compiler designed to simplify the development of robust distributed systems. Developers define their applications using an expressive high-level specification language that hides many of the mundane details associated with implementing distributed applications. The Mace compiler translates the high-level specifications into standard C++ implementations. Mace follows three design principles.

It utilizes service objects to implement individual layers of the system. The interface for each layer specifies functionality provided by the layer as well as requirements for using the layer. The layers in turn are used to construct a complex system's hierarchy.

Mace uses events as unified concurrency model for all levels of the system: within a layer, across layers on a single node, and across the nodes comprising the system. Each event corresponds to a method implemented by a service object. Thus, for example, the transition between layers is done via callbacks that are triggered by events.

Mace uses aspects, computations that cut across the object and event boundaries, to define tasks that need to be performed when certain conditions are met. Aspects are used to cleanly specify how to consistently update local state in response to a variety of cross-layer events, such as node arrivals, departures, and application-level failures.

All of the three principles complement each other and address the complexity and challenges of building robust high performance distributed system while preserving the high-level structure.

Furthermore, much of the code needed for failure detection and handling is automatically generated from semantic information embedded in the system specification. This approach significantly improves code readability and reduces the complexity associated with maintaining the application. Mace's state transition model further enables the practical model checking of distributed systems implementations to find both safety and liveness bugs [18].

Mace is fully operational and has been in development for over four years. Many distributed systems have been built using Mace during this time, including several that involve the creation of robust overlay networks for data storage and dissemination [20]. As a result, Mace provides built-in support for the common operations used in overlay network creation and maintenance.

One particularly useful protocol provided by Mace is RandTree. RandTree is a protocol that constructs an overlay network based on a random tree. Participants in a RandTree overlay network use the RanSub protocol [19] to distribute fixed-size random subsets of global state to all overlay participants, overcoming scaling limitations and improving routing performance. By ensuring that the received subsets are uniformly representative of the entire set of participants and are frequently refreshed, nodes will eventually receive information regarding a large fraction of participants. This allows the RandTree protocol to dynamically adapt to changing network conditions and reconfigure in the face of failures. In short, RandTree creates a random tree with a predefined maximum degree, and automatically handles reconfiguration when any participating node fails, including the root.

2.3 Merging Plush and Mace

In summary, Plush is an application management infrastructure that is designed to simplify distributed application execution. Although it is robust for small clusters, Plush does not provide the scalability or fault tolerance necessary in today's large-scale clusters. Mace is a C++ language extension that simplifies the development of distributed systems and provides built-in support for creating some basic overlay networks, including many overlay trees. Thus, by merging the two systems to create a new system called Plush-M, we can leverage the advantages of both Plush and Mace to create a scalable application management infrastructure that is based on a robust overlay tree provided by Mace.

Plush is designed as a monolithic layered application. To utilize an overlay

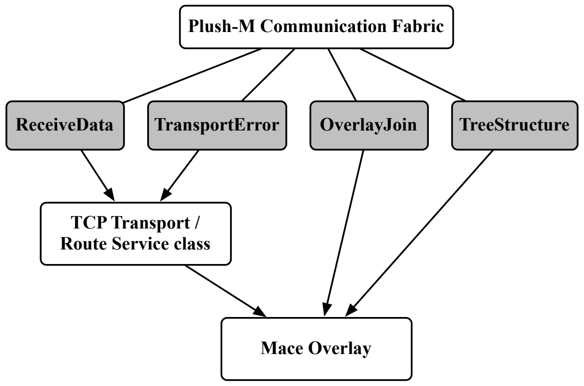
Plush-M Communication Fabric
ReceiveData
TransportError
OverlayJoin
TreeStructure
TCP Transport / Route Service class
Mace Overlay

Figure 2.1: Plush-M communication fabric interacting with a Mace overlay. The gray boxes show part of the API used by Plush-M to interact with Mace. The TCP Transport and Route Service is also provided by Mace.

provided by Mace, the entire communication subsystem in Plush must be replaced. However, we still want to maintain the layer integrity within Plush itself. We accomplish these goals by treating the Mace overlay as a black box underneath Plush, exposing only a simple API for interacting with the overlay network. In the original design of Plush, the controller had to manage the topology of the communication fabric. By inserting a Mace overlay as a black box that is used for communication, the details of the overlay construction and maintenance are hidden from the controller. The Plush controller is mostly unaware of the topology of the hosts that are involved in an execution; the controller knows only the identity of the participating nodes.

This "black box" design not only allows for the independent upgrade of Plush and Mace, but also gives us the ability to use a variety of overlay services and protocols provided by Mace since most use the same API (shown in Figure 2.1). We hope to eventually provide developers with the functionality required to select the overlay network protocol that is best suited to handle the communication needs of their application. It is important to realize that since Mace is a general purpose framework for designing distributed applications, its integration into Plush-M may introduce a slight overhead when compared to the simple tree that Plush provides, but we believe that this is a small price to pay for the improved scalability and fault tolerance.

Although the functionality provided by Mace allows developers to build a variety of overlay networks, we chose to integrate the RandTree protocol into Plush-M. There are several reasons why we chose RandTree instead of other tree-building protocols [15, 13]. In particular, RandTree provides the following benefits:

- **Simple** - RandTree's design is simple and efficient. Since it does not try to optimize for latency or bandwidth like many other overlay tree-building algorithms, the tree can be constructed quickly based on random decisions.
- **Adaptive** - Since RandTree is based on the RanSub protocol, it is adaptive to changing network conditions, which is an important property to have when trying to achieve fault tolerance in potentially volatile conditions.
- **Mature** - The Mace implementation of RandTree is stable and mature, and has been successfully used by many other developers in the past. This minimizes the possibility of bugs and design problems that might occur if we implemented a new protocol from scratch.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Chapter 3

Design and Implementation of Plush-M

This chapter discusses the design and implementation of Plush-M, an extension of Plush that provides improved scalability and fault tolerance by using Mace's RandTree protocol as the underlying communication fabric. We start by describing the message processing, followed by a discussion of the organization of the new tree topology, and then a summary of the overlay construction and disconnection procedures. Lastly, we discuss the addition of a new auto-upgrade mechanism in Plush-M.

3.1 Message Processing

The guiding principle of integrating Mace as the communication layer for Plush-M was to minimize code modifications to the original version of Plush that might lead to logic errors in other areas of application.

Plush's original communication system followed a layered design, as shown in Figure 3.1. At the topmost level, a `Mesh` object was responsible for processing messages, tracking active nodes, maintaining the overlay, and managing `Connection` objects. Each `Connection` object relied on two additional abstractions, called the `Reader` and `Writer`, for message IO. The `Reader` object is responsible for reading, buffering, and parsing messages. Parsed messages are passed to the `Connection` object via a callback. The `Connection` object does minor message processing, such as updating a local timestamp, and in turn, passes the message to the `Mesh` object for

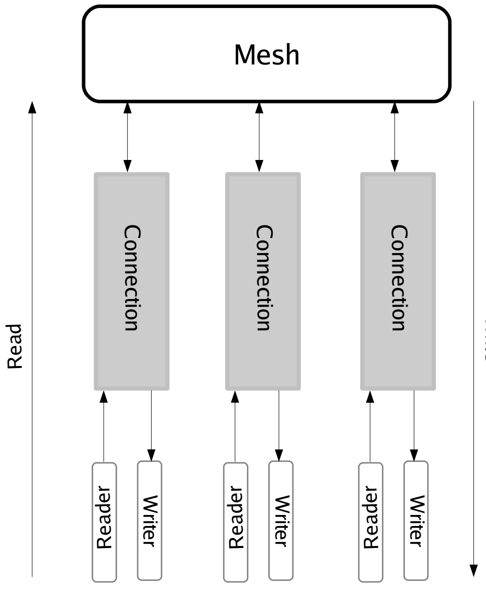

Figure 3.1: Plush's original scheme for message reading/writing.

further processing. The sending procedure follows the same path of control as the read procedure in reverse direction. First, the `Mesh` object sends the message object to the respective `Connection` associated with the host to which the message is addressed. The `Connection` object then passes the message to the registered `Writer` object via a callback. The `Writer` composes the message string and writes it out to a socket.

Mace already provides the necessary mechanisms for sending and receiving messages. To integrate the two systems together, all of the original Plush abstractions above the communication layer are preserved to minimize modifications, but abstractions lower than the `Connection` abstraction are rendered non-functional, with the exception of the string parsing functionality of `Readers`. The `Connection` objects in Plush are, for the most part, preserved with their existing functionality since they are integral for Plush's method of keeping track of remote hosts.

The integrated Plush-M system instantiates Mace's `Transport` object inside a top-level `Mesh` object. (See Figure 2.1). Sending of messages is accomplished via

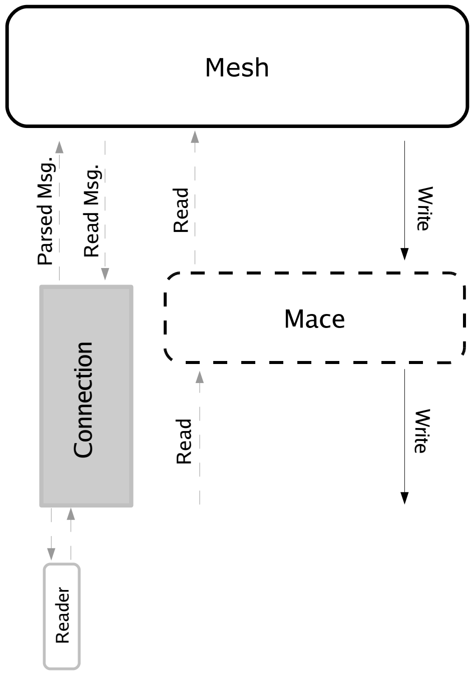

Figure 3.2: Plush-M message IO. Mace handles reading and writing. Connections are preserved to track connected hosts and incoming message parsing.

calls to the appropriate `Transport`'s method and providing messages in the form of strings. For receiving messages, a callback method is registered with the `Transport` object that is called on message reception. On message reception, the `Mesh` examines the list of the existing active `Connection` objects, and if the message is from a host that does not have connection registered, a new one is created. To take advantage of an existing Plush parsing mechanism, the message string is passed to the corresponding `Connection` and `Reader` objects for parsing and processing. The message is then passed back from the `Reader` to the `Connection` object, and then back up to the `Mesh`, following Plush's original processing path. Figure 3.2 illustrates this procedure.

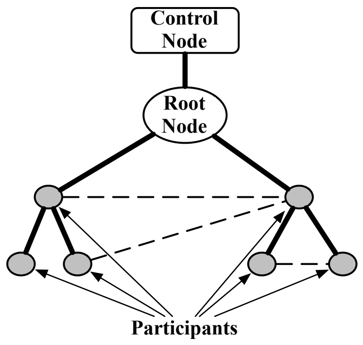

Figure 3.3: Separation of the control and root nodes in Plush-M. Plush-M uses RandTree to build its communication infrastructure. As part of the RandTree protocol, RanSub is used to deliver subsets of global state to all overlay participants. The RanSub links are shown using dotted lines, while the direct tree connections are shown using thick solid lines. RanSub links supplement direct tree links, which increases node interconnectivity and improves message propagation and failure detection.

3.2 Separating Control and Root Nodes

The original Plush design relies on communication via a star or basic tree topology, with the control node always being the center of the star or root of the tree. With the introduction of RandTree as a black box beneath the Plush communication subsystem, the root selection mechanism becomes hidden from Plush. The functionality of the control node, which includes listening for input from the developer and directing the flow of control in executions, must be separated from the root node. In fact, the control node in Plush-M is not even connected to the overlay. Instead, the control node maintains a direct TCP connection to the root of the tree, as shown in Figure 3.3. This complicates the design of Plush-M since additional functionality is needed to keep the control and root nodes informed of each other.

The application developer designates the identity of the control node when Plush-M is started. Typically this node is chosen to be the developer's desktop computer. When managing an application using the original design of Plush, the control node establishes a connection to the required number of computers and invites them to participate in the execution of the application. Since the root of the tree is no longer the same computer as the control node, this process is a bit more complex in Plush-M.

First, the control node randomly selects a root node and sends it an invitation. Once the control node connects to the newly chosen root, the controller sends all subsequent invitations that it creates on to the root node for forwarding. The process of extending an invitation involves first verifying the liveness of the invited node, and then starting the Plush-M client process on it. For large-scale clusters, the number of outstanding invitations can grow quite large. Thus, in order to reduce the load at the root, the controller limits the number of outstanding invitations to a predetermined maximum value.

Through the lifetime of the application, different participant nodes may join and leave the overlay. The root of the tree can also change in Plush-M, which is something that never occurred in the original design of Plush. When changes to the overlay occur, Plush-M receives upcalls from Mace. If the root of the tree changes, Mace notifies the old root node (via an upcall) that it is no longer the root. The old root must make sure the new root knows the identity of the control node. Thus the old root then sends a `SETCTRL` message through the overlay to the new root using a multicast "collect call" provided by Mace. The collect call ensures that the message is successfully delivered to the new root of the overlay. Upon receiving a `SETCTRL` message, the newly elected root node must inform the control node of its identity. To accomplish this, the new root sends a `SETROOT` message to the control node. When the control node receives the `SETROOT` message, it updates its state with the new root's identity, initiates a connection to the new root, and closes its connection to the former root.

Furthermore, Plush-M ensures that the control node stays connected to the acting root node of the overlay, in case some of the control messages get lost in transition. The control node periodically (every two minutes) sends a `SETCTRL` message to the node that it believes is the root of the tree. The `SETCTRL` message is then processed in the usual way, using a multicast collect call to traverse the tree up to the root. The root responds with its `SETROOT` message as usual. If the control node receives a `SETROOT` message from a node other than the one it thought was the root, it updates its state. This same procedure is used to update the control node when it reconnects after a disconnection.

3.3 Control Node Responsibilities

The control node performs several essential actions throughout the duration of the application's execution. First and foremost, it provides a user interface for the application developer to interact with the execution. Secondly, the control node is responsible for tracking outgoing invitations to join the overlay. These invitations are created in response to input from the application developer, or as part of an execution. The control node maintains a queue of records containing the name of each invited node. The record persists until the control node receives a confirmation of the node joining the overlay, or until notice of a failure is received. The invitation queue allows the control node to account for all invitations and limit the number of unconfirmed invitations.

The control node in Plush-M also plays the role of the execution controller and barrier manager, just like the control node in Plush. The control node ensures that all participants in the overlay receive any required application-specific software or data, and if synchronization barriers are specified, ensures the synchronized execution of the application on all participating nodes. The control node can be disconnected if it is not actively inviting nodes or controlling an execution, such as during a long-running computation. Once the control node reconnects, the root tells it the list of participating nodes in the overlay. Unfortunately, in the current implementation of Plush-M, if the control node fails while there are outstanding invitations, all of its active state is lost. In such cases, the control node must reconnect to the root, destroy the existing overlay, and start over. We are currently exploring ways to restore the state of the control node without requiring a complete restart of the existing execution.

3.4 Creating the Overlay Tree

All invitations to join the overlay originate at the control node. Note that the Plush-M controller is responsible for choosing which computers will host the application; we only rely on Mace for managing the communication fabric among participants. Invitations in Plush-M take the form of `SETPEER` messages that are sent from the controller to the root node, which then extends the invitation. SETPEER messages serve two purposes: 1) they test the liveness of the potential participant, and 2) they tell the potential participant to join the overlay. If SETPEER fails, then either the potential participant is offline, or the potential participant is online but not currently running the Plush-M client process. In the latter case, the root node triggers a remote startup mechanism that launches the Plush-M client executable on the potential participant, and then sends the SETPEER message again. If the node is offline or unable to start the Plush-M client process, the root informs the controller of the failure.

Upon receiving a SETPEER message, each potential participant makes a downcall to Mace asking to join the overlay. Making the remote node responsible for initiating the joining of the overlay elegantly avoids complications presented by alternative approaches. If the invitation is triggered from the root, the node is immediately added to the root's internal list of participants in Mace. In addition, Mace periodically (every 10 seconds) tests all participants for liveness. Note that there is a possibility of receiving a TCP error notification for a node being invited, especially if there is a delay in the time it takes the Plush-M client to start. Such scenario unnecessarily complicates the invitation process.

Once the node has successfully joined, Mace notifies the parent of the newly joined node via the peerJoinedOverlay upcall in Plush-M. Once this upcall completes, the connection has been established, and the job of the Mace subsystem in the join procedure is complete. The parent node then informs the new child of any relevant Plush-M specific details. The parent sends SETROOT, INVITE, and SETCTRL messages to the child to inform it of the identity of the controller and root nodes. The child node processes these messages, and responds to the INVITE by sending a JOIN message to the root. The root node processes the JOIN message and adds the newly joined host to the table of participating nodes, completing Plush's portion of the invitation.

Alternatively, if the node takes longer than five minutes to be invited without completing the invitation cycle it is marked as failed by Plush-M and the connection to it is terminated.

3.4.1 Connection Maintenance in Plush-M

The original design of Plush uses “connection” objects internally to help keep track of the connection status of participating hosts. Retaining this functionality in Plush-M required the addition of several new Mace upcall methods. Mace calls the `notifyParent` method on overlay participants every time there is a change in a parent node. This method provides the identity of the new parent. It creates a new `Connection` object to track the new parent host and deactivates the old parent’s `Connection` object by marking it as `MOVED`. The `MOVED` status indicates that the two hosts are no longer directly connected in the overlay. Reports of any errors on connections that are marked as `MOVED` are verified rather than being ignored if no `Connection` object exists for the node, or reported as errors if the connection is marked as active. This is explained in more detail in Section 3.5.

The `notifyChildren` method is called on overlay participants whenever a child of the participant changes. It provides the identities of all actively connected children. The method compares the new list of children to the existing list of connections and performs similar bookkeeping as `notifyParent` by marking some `Connection` objects as `MOVED` and creating new ones if necessary.

The `notifyIsRootChanged` method is called at the new and former root when the root is switched. This causes the former root to remove its connection to the control node, if one exists, and send a `SETCTRL` message via a `collect` call. This message eventually reaches the new root, notifies it of the control node’s identity, and triggers a `SETROOT` message to be sent from the new root to the control node.

All three of these methods help keep Plush-M’s internal host and connection structures current, by invalidating old connections and creating new ones as needed. This, in turn, allows for better error handling.

3.4.2 Connection Maintenance in Mace

The inner workings of Mace connection handling are hidden from Plush-M and the interface that Mace exposes to Plush-M is relatively limited in scope. This presented a problem as we tried to construct topologies with more than 500 nodes. As previously mentioned, all invitations are handled from the root, which entails opening a socket to send messages to remote hosts. As he number of connections grows, a file descriptor limit is reached by the root node. This, in turn, triggers Mace's garbage collection mechanism which closes half of all connections based on the LRU (least recently used) principle. In some cases, Mace closes the connection to the control node thus forcing a reconnection, but this also leads to a loss of control messages, and as a result, a degradation of performance. To prevent the control node's connection from going stale, the root sends a no-op keep-alive message to the control node. This is a temporary solution and we are currently exploring ways to expose more of Mace's interface to prevent certain important connections from being garbage-collected.

3.5 Overlay Error Detection

Similar to the upcalls for overlay maintenance, Mace reports network communication errors in the communication fabric through another set of upcalls. Plush-M relies on this mechanism to track the node connection status for all participants in the overlay. When a node receives an upcall from Mace indicating a communication error to another host, the receiving node reacts by deleting the connection object for the disconnected host. Then, to ensure any necessary tree reconfigurations are initiated, the receiving node checks the disconnected host's position in the overlay tree relative to itself. If the disconnected host is the node's child or parent in the tree, a `DISCONNECTED` message is forwarded to the root node. Alternatively, if the disconnected host is neither a parent or child of the node, the node forwards a `GETSTATUS` message up to the root. The root in turn sends a `GETSTATUS` message to the host in question. If the root node confirms the disconnection, the root marks the disconnected host as `DISCONNECTED`, informs the control node, and tree reconfiguration begins.

During an application's execution, some nodes might experience network outages and be reported as `DISCONNECTED` to the root and control nodes. Depending on the length of the execution, the control node may later try to re-establish connection to the `DISCONNECTED` node(s). If the reconnection is unsuccessful, the disconnected node is marked as `FAILED` and is permanently removed from the execution. If the reconnection is successful, the node is reincorporated into the execution. However, if this node continues to disconnect frequently, it is assumed to have an unreliable network connection. After a preset number of disconnects within a specified timeframe, the node is marked as `FAILED` and is never re-invited to participate.

3.6 Internal Name Reconciliation

The original design of Plush relies on a structure of type `HostID` to record hostname and port usage information about participating nodes. Similarly, Mace relies on objects of type `MaceKey` to identify its nodes, which, in our case, also contains the node's IP address. When integrating Plush and Mace in the implementation of Plush-M, a problem arose when nodes had multiple aliased hostnames. When a node receives a message from another participant in Plush-M, the receiving node initializes a `HostID` object with a hostname derived from `MaceKey`'s IP address. If the node uses an aliased hostname, the resolution process may produce a different hostname than the name already registered as a participant. This in turn caused messages to be rejected, since the resulting hostname is not registered as a valid host. This caused many problems in our initial implementation of Plush-M.

To address the problems involving conflicting hostnames and ensure the proper operation of Plush-M, we had to reconcile the two addressing systems. The first step in this reconciliation involved adding a `MaceKey` object as a member of the `HostID` structure. Since Plush-M extensively relies on `HostID` objects throughout the duration of the execution, many additional changes were also required. Plush only resolves hostnames into IP addresses when a new connection is initialized to a new participant. Following the same principle in Plush-M, the `MaceKey` object is initialized, which includes resolving the hostname into an IP address, for all nodes that are invited to participate in the overlay. Initializing a `MaceKey` object for all available cluster nodes is expensive and wasteful, especially if only a fraction of them are required for execution. Thus `MaceKey` objects are only initialized for nodes that are involved in an execution. One caveat of this approach is ensuring that `HostID`s with initialized `MaceKey`s are never compared to `HostID`s without initialized `MaceKey`s in the Plush-M implementation. (This can only occur on the control node.)

3.7 Disconnecting the Overlay

Once the execution completes, the overlay must be dismantled before the control node disconnects from the root. Failure to do so will result in the overlay continuing to exist even after the control node disconnects. While there is nothing inherently wrong with keeping the overlay connected, it does waste computing resources. To disassemble the overlay, a disconnect command is issued by the control node. The disconnect command causes a `DISCONNECT` message to be sent from the control node to the root node, and then from the root node to all other nodes in the overlay via a broadcast mechanism provided by Mace. When a node receives the `DISCONNECT` message, it attempts to terminate the execution of the Plush-M client process. The termination of the process is allowed only if the node has no connected children. Thus, the root node must wait for all of its children to disconnect, and hence the root always disconnects last. Furthermore, once a node receives the `DISCONNECT` message, it broadcasts additional `DISCONNECT` messages repeatedly to its children until all children have disconnected. This approach works even in the face of any overlay reconfiguration that occurs as the nodes start leaving the overlay.

To further speed-up the process of overlay dismantling, each node in the overlay starts a two minute timer upon receiving a new `DISCONNECT` message. When the timer expires the Plush-M client terminates, even if the node still has connected children. This scenario, and some other situations involving uncommon errors, has the potential to leave individual unconnected nodes in an active state running the Plush-M client. These unattended nodes eventually detect that they have no active connections left, causing them to set their two minute expiration timer. If the node does not receive any messages or start any new connections within the expiration time, the Plush-M client will be terminated. Note that the process of overlay dismantling on large topologies and trees of greater maximum depths does take longer due to the increased number of network hops between the root node and the leaf participants. However, with the disconnection timeout mechanisms described above, the control node is safe to terminate without waiting for hosts to report their disconnection.

3.8 Client Auto Upgrade

During the development of Plush-M it was essential to have the latest version of the client present on participating nodes for proper operation. We introduced a new automatic upgrade mechanism in Plush-M to support this functionality. Since managing software versions and supporting upgrades are both common actions in a distributed system, our initial intention was to make this a general abstraction in Plush-M that developers could use to simplify node configuration and application management. The automatic client upgrade mechanism in Plush-M is quite simple. Prior to starting the Plush-M client on each participating node, Plush-M runs a maintenance script on the node that obtains versioning information for the latest Plush-M client from an external software repository. If the node is running an outdated version of the Plush-M client, the maintenance script downloads and installs the latest client from the repository. The concurrent invitation limit, described in Section 3.2, prevents flooding of the upgrade repository with requests in this situation. The maintenance script is also capable of performing other tasks, such as removing old data or log files. The upgrade script is copied from the control node to the root on startup, and then from the root to other nodes joining the overlay. This, in turn, allows for customization of upgrade script's functionality.

On long running overlays with a rolling upgrade, which gradually updates each host's Plush-M client, we update the entire overlay. By terminating Plush-M clients on these hosts, the node recovery procedure will execute the upgrade script, bringing upgraded client online. In this case, the order of nodes being terminated must be centrally controlled since there is no upgrade scheduling functions [1] on hosts themselves.

Although the use of the maintenance script and the automatic client upgrade mechanism works as desired and accomplishes our goals, we find that the upgrade procedure can be quite expensive, especially if the node has a slow or lossy network connection to the software repository. One possible way to alleviate this problem is to have several upgrade repositories, that might provide better latencies to different geographical locations. Since the upgrade script is distributed as part of the upgrade mechanism the list of upgrade repositories can be easily modified. In addition, our experience shows that it is important to provide a way to disable the automatic client upgrade mechanism, since clusters that run a common file system do not need to download updates. We do believe that automating software upgrades is an important aspect of application management, however further investigation and experimentation is required to improve performance. We revisit the performance of the automatic upgrade feature in more detail in Chapter 4.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Chapter 4

Evaluation

Since our overall goal was to improve the scalability and fault tolerance of Plush, to evaluate our work we tested Plush-M's ability to connect to several hundred computing resources. As mentioned previously, Plush-M utilized Mace's RandTree overlay that provided the needed scalability and fault tolerance. We performed experiments to measure scalability and fault tolerance in wide-area environments using PlanetLab, and in emulated environments using ModelNet. For our PlanetLab experiments, we used approximately 100 randomly chosen PlanetLab hosts. Although PlanetLab contains almost 1000 hosts, we found it difficult to find more than 100 that were usable and available to host our experiment at any point in time. (Note that this limitation is due to the underlying infrastructure of PlanetLab, and has nothing to do with Plush-M.) To further test scalability, we created emulated ModelNet topologies with varying numbers of virtual clients using 17 physical machines. For ModelNet topologies with more than 1000 virtual clients, the physical machines began to run out of memory during experiments. Thus, our testing was limited to topologies with 1000 virtual clients, but we are confident that Plush-M is capable of scaling and recovering from failures in even larger topologies.

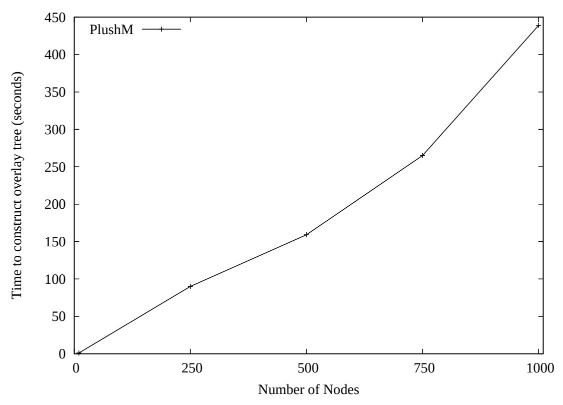

Figure 4.1: Time to construct RandTree in Plush-M using varying number of virtual ModelNet clients. Maximum of 12 children per node. Emulated on 17 physical machines.

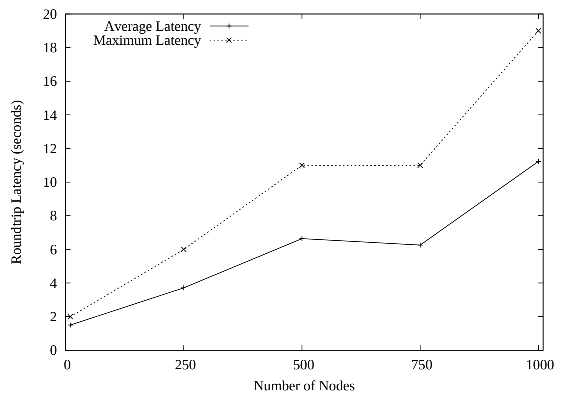

Figure 4.2: Average and maximum latency required for a message to be sent from the Plush-M controller to all participants, and for all participants to send a reply. Maximum of 12 children per node. Emulated on 17 physical machines.

4.1 Scalability

Our first set of experiments tested the scalability of Plush-M for varying numbers of ModelNet emulated clients. The goal was to show that Plush-M scales beyond Plush's limit, which is approximately 300 nodes. To verify Plush-M's scalability, we measured the completion time for the following two operations: 1) the time to construct the overlay tree, and 2) the roundtrip time for a message sent from the control nodes to all participants and then back to the control node. We ran these experiments for 10, 250, 500, 750, and 1000 clients. The results are shown in Figure 4.1 and Figure 4.2. In most of experiments, we limited the construction of RandTree to allow a maximum of twelve children per node, unless specified otherwise. (Note that the number of children was picked somewhat arbitrarily. Twelve is simply the default value in the RandTree library.) Therefore, for a perfectly balanced tree, the depth would be approximately 3 for all experiments except where the number of clients is 10. In addition, we limited the number of concurrent outstanding invitations to 50. The length of time required for the control node to connect to the root of the overlay is not included in our measurements. Also note that the auto-upgrade mechanism was disabled during these experiments.

The overlay tree construction times measured (and shown in Figure 4.1) were longer than we expected them to be. We speculate that this is due to the start-up time of the client on the participating nodes. Preliminary testing revealed that increasing the maximum size of the outstanding invitation queue from 50 to 100 invitations did not have a significant effect on the connection time. Similarly, changing the maximum out-degree of the nodes from 12 to 50 also did not have a noticeable effect. Further investigation of the issue indicated that the problem resulted from a limitation in our experimental setup. Since multiple ModelNet clients are emulated on a single machine, there is a limit on how many processes the machine can start simultaneously, and thus some clients experience a delay when starting the Plush-M process. We believe that the construction time would be much faster if more machines were available for experimentation.

To test the message propagation time in the overlay, we first waited until the tree was completely constructed. Then we broadcast a message from the control node out to all participants and waited for a reply. We measured the time between when we sent the message until each node replied. The broadcast message triggered the execution of the ‘hostname’ command on each remote node. By default, all terminal output is sent back up the tree to the root and then passed to the control node. Figure 4.2 shows our results. One line indicates the maximum time measured, and the other line indicates the average time across all participants for the reply message to be received by the control node. In summary, we were relatively pleased with the measured times, since

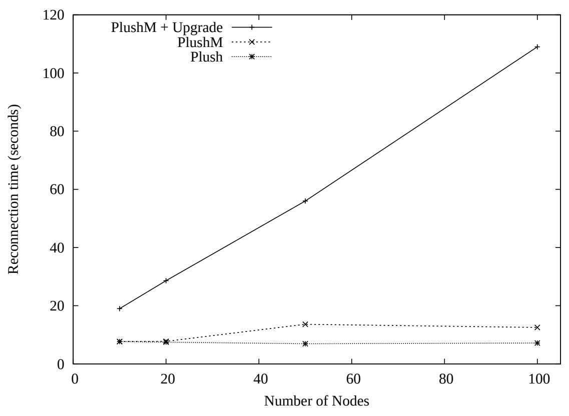

Figure 4.3: Average recovery time of PlanetLab nodes, after failure of 25% of the overlay participants.

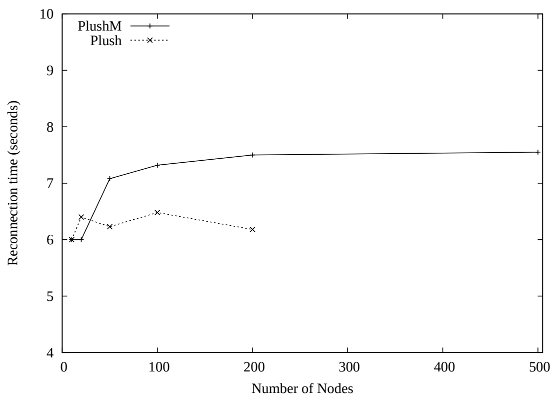

Figure 4.4: Average recovery time of ModelNet nodes, after failure of 25% of the overlay participants.

we were able to receive replies from most hosts in less than 10 seconds (on average) for our 1000-node topology. Since the entire topology was emulated on 17 machines, 1000 nodes resulted in 59 concurrent clients running on single physical machine, which explains the increase in the roundtrip time. Note that in this experiment we were unable to compare to Plush, since Plush does not support execution on more than 300 nodes.

4.2 Fault Tolerance

Providing fault tolerance in Plush-M largely depends on being able to detect and recover from failures quickly. Thus, we conducted several experiments to measure the failure recovery time for participants in the overlay when a host-level failure occurred. The goal was to see how long it took for the failed nodes to rejoin the overlay tree after being manually disconnected to simulate failure. By using RandTree instead of the default star in Plush, Plush-M incurs some overhead associated with tree communication and reconfiguration. We wanted to verify that this overhead did not negatively impact Plush-M's ability to detect and recovery from failures quickly.

To test failure recovery and fault tolerance in Plush-M, first we started an application on all nodes that just executed "sleep" for a sufficiently long period of time. During the execution of the sleep command, one quarter of the participating nodes were failed, which essentially amounted to manually killing the Plush-M client process. We then measured the time required for the failed hosts to detect the failure, restart Plush-M, and rejoin the overlay (possibly causing a tree reconfiguration). We repeated the experiment with varying numbers of nodes to explore how the total number of participants affected the recovery time. Plush-M was again run with RandTree using a maximum of twelve children per node.

4.2.1 Failure Recovery on PlanetLab

Figure 4.3 shows the results from running our failure recovery experiment on PlanetLab with the number of hosts ranging from 10 to 100. As previously mentioned, it took a prohibitively long time to find more than 100 working PlanetLab nodes. The results show the average reconnection time of Plush-M running with automatic upgrade enabled, Plush-M running without automatic upgrade, and Plush executing over its default star topology. We could not compare to Plush with the simple tree topology, since failure recovery that involves a tree reconfiguration is not supported. As Figure 4.3 shows, automatic upgrade in Plush-M was a major contributor to the latency required for node reconnection. This was largely due to the fact that the failed clients contacted the software repository to verify that they had the latest Plush-M client installed (as part of the automatic upgrade mechanism) before rejoining the overlay.

Figure 4.3 also shows that with automatic upgrade enabled, as the number of participants increased, so did the average reconnection time. This was expected, since many PlanetLab nodes were running with high CPU loads and had slow connectivity to the software version server. With automatic upgrade disabled, the results are much more promising. The average reconnection latency using Plush-M and Plush was very close. For Plush-M, the average reconnection time for overlays with more than 12 participants took approximately 12 seconds. When less than 12 nodes were involved, RandTree created a tree with one level and all nodes directly linking to the root. In this case the average time was about 8 seconds. Plush running over its default star topology performed consistently as the number of nodes increased, with an average reconnect time of about 7 seconds. The difference between Plush-M and Plush on larger overlays is likely caused by the overhead of tree reconfiguration. However, this overhead was relatively small (less than 7 seconds for all experiments), and did not seem to increase significantly as the number of participants grew.

4.2.2 Failure Recovery on ModelNet

In Figure 4.4, we performed the same failure recovery experiment on ModelNet. The goal again was to measure how quickly Plush-M recovered and reconfigured after a failure was detected. Note that we did not repeat the automatic upgrade experiment on ModelNet, since Plush-M ran from a shared partition mounted on all emulated hosts. A version check would be pointless in this case. Our results show that the reconnection latency difference between Plush-M and Plush is less than 2 seconds in most cases, and can again be explained by the overhead associated with the tree reconfiguration during node reconnections. Since variable wide-area network conditions did not affect our results, the overhead of tree reconfiguration on ModelNet is must less significant than the overhead of tree configuration on PlanetLab. This result bodes well for other large-scale cluster environments that are not spread across the wide-area.

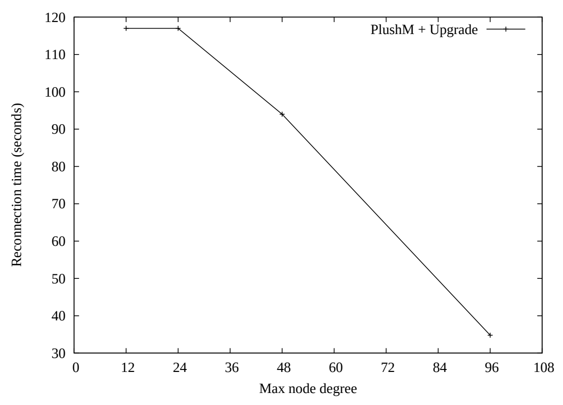

Figure 4.5: Effect of RandTree node degree on average reconnection time after failure on PlanetLab.

4.2.3 Tree-reconfiguration on PlanetLab

For our final experiment, we wanted to see how the structure of the tree affected the average reconnection latency of a node. To test this, we ran a set of experiments similar to the ones previously described. First, we used Plush-M with the automatic upgrade feature enabled to run "sleep" on PlanetLab using 100 overlay participants. Then we killed the Plush-M clients on 25 participants. The overlay detected the failure and tried to reconnect the disconnected nodes. We measured the average time of node reconnection, as before. We repeated this experiment several times, varying the maximum out-degree allowed in RandTree from 12 to 96. The results are shown in Figure 4.5. As expected, the average reconnection time decreased as the degree increased. This is due to the fact that increasing the maximum degree allowed decreased the depth of the overlay tree, and simplified tree reconfiguration. Nevertheless, compared to the previous results, the reconnection time was still dominated by the automatic upgrade mechanism.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Chapter 5

Related Work

Since Plush-M is an extension of Plush, much of the work that is related to Plush is also related to Plush-M. In this section we revisit some of these projects, focusing the discussion on fault tolerance and scalability.

In the context of remote job execution, Plush-M has similar goals as cfengine [8] and gexec [10]. However, since Plush-M can be used to actively monitor and manage a distributed application, and also supports automatic failure recovery and reconfiguration, the functionality provided by cfengine and gexec is only a subset of the functionality provided by Plush-M. Like Plush, cfengine defaults to using a star topology for communication, with additional configuration support for constructing custom topologies. This constructed topology typically reflects the administrative needs of the network, and depending on what topology is used, can impact scalability. gexec relies on building an n-ary tree of TCP sockets, and propagates control information up and down the tree. The topology allows gexec to scale to over 1000 nodes, and seems similar to the approach used in Plush-M.

SmartFrog [12] and the PlanetLab Application Manager (appmanager) [14] are designed to manage distributed applications. SmartFrog is a framework for building and deploying distributed applications. SmartFrog daemons running on each participating node work together to manage distributed applications. Unlike Plush-M, there is no central point of control in SmartFrog; work-flows are fully distributed and decentralized. However, communication in SmartFrog is based on Java RMI, which can become a bottleneck for large numbers of participants and long-lived connections. appmanager is designed solely for managing long-running PlanetLab services. It does not support persistent connections among participants. Since it utilizes a simple client-server model, it most closely resembles the Plush star topology, and has similar scalability limitations. The scalability also largely depends on the capacity of the server and the number of simultaneous requests.

Condor [7] is a workload management system for compute-bound jobs. Condor takes advantage of under-utilized cycles on machines within an organization for hosting distributed executions. The communication topology of Condor uses a central manager which is directly connected to the machines in the resource pool, constructing a star topology like Plush. The scale of this design is limited by the constraints of the operating system, such as file descriptor limits, similar to the star topology in the original design of Plush.

Remote Maintenance Shell (RMS) [16] is a method for service management and maintenance in grid environments. Its basic functionality provides migration, installation, restart, and basic version handling for services adapted to work with RMS. RMS consists of three components: Central Management Station, Mobile Agent, and Maintenance Environment. Management Station initiates the maintenance process, stores necessary data in the Mobile agent, and starts their distribution. Mobile Agent migrates from one host environment to another executing the maintenance process. Maintenance Environment provides a framework for maintenance process execution and is pre-installed on each host. Mobile Agents enable RMS to perform decentralized, asynchronous operation execution. Plush's upgrade is similar to RMS in the respect that the upgrade script resembles mobile agent in its functionality, and the fact that it is migrated from node to node. Nevertheless, Plush-M relies on centralized repository to access the upgrades, rather than distributing them along with upgrade script. Furthermore, Plush-M's upgrade mechanism is much less ambitious in scope and simpler in implementation, since it primarily handles the upgrade of only one service, which is Plush-M itself.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Chapter 6

Future Work

There are several aspects of the design of Plush-M that warrant further exploration. Even though our experiments were limited to 1000 nodes, we believe that our current design has the ability to scale to thousands of machines, which is required in today's large-scale computing platforms. However we are still somewhat dissatisfied with the initial connection time for our overlay tree. We hope that the heightened times are due to limitations in our testing infrastructure rather than in the design of our system. One of the disadvantages of using Plush and Mace is that both systems attempt to hide the underlying complexities of distributed applications from developers. In this case, it would ease debugging if some of the underlying mechanisms were exposed.

In addition, we plan to evaluate the scalability and fault tolerance of Plush-M on much larger topologies. Thus far we have been limited by the unavailability of physical cluster machines and PlanetLab resources to host our experiments. We plan to gain access to larger testbeds and perform more extensive testing. While we are confident that our design will scale to much larger topologies, we need to verify correct operation in a realistic large-scale computing environment. This will also allow us to identify additional performance bottlenecks that only appear when large numbers of participants are present.

Furthermore, treating Mace as a black box posed some limitations. As discussed in Section 3.4.2 some of the essential connections were garbage collected by

Mace's connection management system. The workaround of keepalive messages might not work all the time. Thus, a more exposed API that provides finer control on connections would be a great improvement.

One last problem that we hope to address relates to how failed nodes are handled in Plush-M and Mace. Currently, when a node fails repeatedly, Plush-M detects that the node may be problematic and marks it as failed. This prevents the control node from attempting to use the node for future computations. However, Plush-M lacks the facility to indicate the failure to Mace. Hence, Mace continually attempts to reestablish the connection with the failed node during every recovery cycle. By introducing an extension to Mace that allows Plush-M to mark certain nodes as failed, we can significantly decrease the traffic associated with overlay maintenance, thus improving performance.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Chapter 7

Conclusions

Plush is an extensible distributed application management infrastructure that does not provide adequate scalability or fault tolerance for large-scale computing platforms due to the design of its underlying communication fabric. We sought to correct these issues and allow for greater scalability and robustness without significantly sacrificing functionality and performance. The result is Plush-M, a modified version of Plush, which implements an interface to Mace communication services and replaces the Plush communication fabric with a tree-based overlay network called RandTree. RandTree is built using Mace, a C++ language extension and source to source compiler for building high-performance distributed systems, and exposes a standard application programming interface. Even though we utilized only one of the Mace's overlays, Plush-M is capable of taking advantage of other overlays designed in Mace that expose a similar application programming interface.

Based on our experimentation thus far with the current overlay, we believe that RandTree is a robust tree-based overlay network that allows Plush-M to scale far beyond the limits of the original Plush design. We believe that the performance bottlenecks experienced are largely due to limitations in our testing infrastructure. Further testing will allow us to confirm these speculations. In addition, RandTree provides Plush-M with advanced failure recovery options, including automatic failure detection and tree reconfiguration. As a result, Plush-M provides a scalable and fault tolerant solution for distributed application management in large-scale computing environments.

This Chapter, in part, is a reprint of the material as it appears in the proceedings of USENIX Workshop on Large-Scale Computing, 2008, Topilski, Nikolay; Albrecht, Jeannie; Vahdat, Amin. The Thesis author was the primary investigator and author of this paper.

Bibliography

[1] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

[2] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2007.

[3] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose Synchronization for Large-Scale Networked Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.

[4] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. PlanetLab Application Management Using Plush. *ACM Operating Systems Review (OSR)*, 40(1), 2006.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2003.

[6] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[7] Allan Bricker, Michael Litzkow, and Miron Livny. Condor Technical Summary. Technical Report 1069, University of Wisconsin–Madison, CS Department, 1991.

[8] Mark Burgess. Cfengine: A Site Configuration Engine. *USENIX Computing Systems*, 8(3), 1995.

[9] Charlie Catlett. The Philosophy of TeraGrid: Building an Open, Extensible, Distributed TeraScale Facility. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2002.

[10] Brent Chun. gexec. http://www.theether.org/gexec/.

[11] Simson Garfinkel. Commodity Grid and Computing with Amazon's S3 and EC2. *;login: (The USENIX Magazine)*, Febuary 2007.

[12] Patrick Goldsack, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, and Peter Toft. SmartFrog: Configuration and Automatic Ignition of Distributed Applications. In *HP Openview University Association Conference (HP OVUA)*, 2003.

[13] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, 2001.

[14] Ryan Huebsch. PlanetLab Application Manager. http://appmanager.berkeley.intel-research.net.

[15] John Jannotti, David K. Gifford, Kirk L. Johnson, Frans M. Kaashoek, and James W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[16] Gordon Jezic, Mario Kusek amd Tomislav Marenic, Ignac Lovrek, Sasa Desic, Krunoslav Trzec, and Bjorn Dellas. Grid Service Managemant by Using Remote Maintenance Shell. In *Proceedings of the International Conference on Grid Services Engeneering and Management, (GSEM 2004)*, 2004.

[17] Charles Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of Programming Languages Design and Implementation (PLDI)*, 2007.

[18] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[19] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[20] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, 2003.

[21] Larry L. Peterson, Andy C. Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences Building PlanetLab. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[22] Amin Vahdat, Ken Yocum, Kesvin Walsh, Priya Mahadevan, Dejan Kostić, Jeffrey Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2002.