UNIVERSITY OF CALIFORNIA, SAN DIEGO

LIVE DEBUGGING OF DISTRIBUTED SYSTEMS

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

by

Darren Duc Dao

Committee in charge:

Professor Amin Vahdat, Chair
Professor Alex C. Snoeren
Professor Alin Deutsch
Professor Jeannie Albrecht

2008

The Thesis of Darren Duc Dao is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____

Chair

University of California, San Diego

2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deep and sincere gratitude to my advisor, Professor Amin Vahdat, for his guidance and support throughout my graduate career. It has been a great privilege to work with him.

Special thanks to Professor Jeannie Albrecht, for being a constant source of motivation. When the obstacles seemed so hard to overcome, she was the one who helped keep my head up.

I also want to thank Professor Alex Snoeren and Professor Alin Deutsch, for graciously being a part of my thesis committee, and for reviewing my thesis.

Finally, I am grateful for my mentor, Charles Killian, for putting aside time in his busy schedule to help me with my thesis project.

VITA

2008	Master of Science in Computer Science University of California, San Diego San Diego, CA
2006	Bachelor of Science in Computer Science University of California, San Diego San Diego, CA

PUBLICATIONS

J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: Distributed application configuration, management, and visualization with plush. In *LISA'07: Proceedings of the 21st conference on 21st Large Installation System Administration Conference*, pages 1–19, Berkeley, CA, USA, 2007. USENIX Association.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Systems and Networking
Professor Amin Vahdat

ABSTRACT OF THE THESIS

LIVE DEBUGGING OF DISTRIBUTED SYSTEMS

by

Darren Duc Dao
Master of Science in Computer Science
University of California, San Diego, 2008
Professor Amin Vahdat, Chair

Debugging distributed systems is a challenging task. The challenge stems from the fact that many errors do not manifest themselves until the systems are deployed to production. Unfortunately, once the systems are deployed, there is no easy way to debug them. In this thesis, I present MaceODB, a tool whose purpose is to assist the programmers in such debugging task.

To use MaceODB, the programmers specify properties for their systems. At runtime, MaceODB uses these properties to check the status of the systems that are running, and reports back to the programmers if there were any errors. Using MaceODB, we were able to detect non-trivial bugs in existing systems. This accomplishment is impressive considering the fact that most of these systems have been tested extensively in the past. More importantly, the results from our macro-benchmarks indicate that MaceODB is low in overhead, thus making it possible to be left running on deployed systems with low performance impact.

Chapter 1

Introduction

Debugging distributed systems has long been a challenging task. This challenge is due to the nature of the environment under which the systems run. Typically, these systems consist of hundreds, if not thousands, of nodes that span across the country. These nodes are often connected by a fragile network, with varying speed and capacity. With these systems, correctness of operation is often a function of not only a single node's behavior, but of the system as a whole. To make the debugging task more challenging, many errors do not manifest themselves until the systems are deployed to production. These errors typically show up only after a certain sequence of distributed events, such as machine and network failures.

To help programmers with the debugging process, many tools have been proposed. However, each has its own drawbacks. For example, with model checking, programmers can define specifications for their systems, and use the model checker to systematically explore the system's state space while looking for bugs. The problem with this approach is that the checks are typically done in a virtualized environment, not in a real live system. Another approach is to use replay-based checking tools. These tools allow programmers to perform careful offline analysis of their systems. However, the drawback of this approach is that the checks are not done at runtime. Finally, with log-based analysis tools, programmers have the ability to systematically process logged outputs while looking for errors. The drawback of this approach is the high cost of

generating and storing the necessary data for the logs.

Discouraged by these drawbacks, most programmers settle for an ad-hoc way of debugging their systems. For example, many programmers end up inserting `printf` statements into their code, then, later on, parsing through the logs, looking for errors and discrepancies. The problem with this approach is that it requires the programmers to know ahead of time what they want to print and what to expect in the logged outputs. This requirement is problematic since it requires the programmers to already know the kinds of bugs that are hidden in the system. Presented with these issues, we were motivated to create a better debugging tool that meets the following requirements:

- Ease of use - Programmers tend to avoid tools that are difficult to use. Therefore, in order for our tool to achieve a high adoption rate, we need to make the tool as easy to use as possible. We strive to hide low-level implementation details from the programmers and automate as many tasks as possible so that the programmers only have to write a minimal amount of additional code.
- Flexible and powerful - Besides making the tool easy to use, we also want to make it powerful and flexible enough to assist programmers in finding a wide variety of bugs.
- Low overhead - Many bugs do not manifest themselves until the systems are deployed to production. In order to catch these bugs, we need to have the tool running on the deployed systems. However, in doing so, the tool might interfere with the systems' performance. Therefore, it is important to make the tool as low in overhead as possible. That way, it can operate without incurring too much negative impact on the systems under test.
- Fault tolerance - Distributed systems typically run on networks that are fragile and unreliable. As a result, the tool needs to be fault-tolerant so that it can continue with the debugging process even when facing problems such as machine or network failures.

Using these requirements as our guidelines, we designed and built MaceODB,

an on-line debugging tool for the Mace [5] language. Using MaceODB, we were able to find non-trivial bugs in existing Mace services, and our performance evaluation shows that MaceODB has very little impact on the applications under test.

The rest of this thesis is organized as follows. Chapter 2 covers the background of Mace. Chapter 3 explains our approach to design MaceODB in order to satisfy the requirements previously presented. Chapter 4 describes the actual implementation. Chapter 5 and 6 report on our experience on using MaceODB and on its performance. Related works are described in Chapter 7. Future work and conclusions are discussed in Chapter 8.

Chapter 2

Background

Since MaceODB is designed specifically for Mace, an overview description of Mace is necessary. That information is covered in Section 2.1. For more detailed information, please refer to the actual Mace paper [5]. In this chapter, we also provide background on the concept of using properties for debugging distributed systems. This concept is important to MaceODB since it serves as the fundamental buidling block for our design and implementation.

2.1 Mace

Mace is a C++ language extension and source-to-source compiler that translates a concise but expressive distributed system specification into a C++ implementation. Mace overcomes the limitations of low-level languages by providing a unified framework for networking and event handling, and the limitations of high-level languages by allowing programmers to write program components in a controlled and structured manner in C++. By imposing structure and restrictions on how applications can be written, Mace supports debugging at a higher level, including support for efficient model checking and causal-path debugging via the Mace model checker [6]. The limitation of the Mace model checker is its inability to debug live systems. MaceODB attempts to solve that limitation.

2.2 Properties

To use MaceODB, programmers write properties for the system that they wish to test. These properties are predicates that must hold true for some particular nodes in the system, or for the system as a whole. There are two types of properties: liveness properties and safety properties.

Safety properties - are properties that should always be true. These properties assert that something bad will never happen, that is, the program will never enter an unacceptable state [7]. Formally, they can be expressed as statements of the form *always p*, where *p* is predicate that must be evaluated to true. An example of a safety property is that in a system that sets up an overlay tree, there should always be no loops.

Liveness properties - are properties that should eventually be true. These properties assert that a program will eventually enter a desirable state, or that something good will eventually happen [7]. For example, in an overlay tree, all nodes should eventually be in the `joined` state. It is important to note that liveness properties, unlike safety properties, apply over the entire program execution, rather than individual states. This makes it harder to evaluate liveness properties. However, the benefit of using liveness over safety is that it provides a more flexible and natural way for checking the system. With safety properties, the programmer must know exactly what violations to check. This requirement is problematic and is similar to the drawback of using `printf` for debugging the system. The problem here is that the programmer must have prior knowledge on the nature of the bugs that might exist in the code. As for liveness properties, the programmer is only required to specify high-level desirable system behaviors. These system behaviors should be easy to specify since they correspond directly to the expected behaviors that are set forth by the system specifications, which is something that should be readily available to the programmer.

Chapter 3

Design

MaceODB is an extension to the Mace compiler. It contains instructions for translating properties into code that will perform property checking at runtime. To use it, the programmer begins by adding properties to the Mace service that he or she wants to debug (see Figure 3.1). After specifying the properties, the next step is to use the Mace compiler to generate C++ code for the service. During this step, if MaceODB is enabled, the Mace compiler will invoke MaceODB to parse through the properties, and generate the additional code that is needed for evaluating the properties listed in the Mace service. The additional code is generated together with the rest of the other code, which will then be compiled into a single executable. At runtime, this executable will periodically invoke the appropriate methods for evaluating the properties, and report to

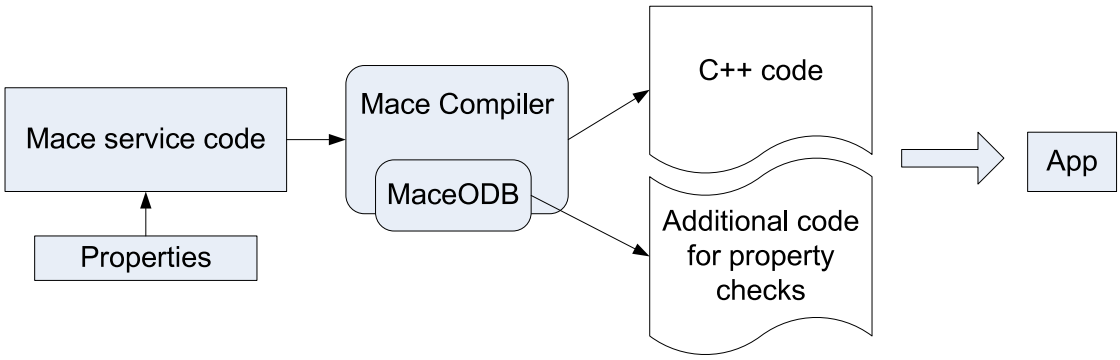


Figure 3.1: Overview of MaceODB

the programmer if there are any failures.

The rest of this section describes the design of MaceODB in more details. Section 3.1 describes the property grammar. Section 3.2 describes the design for evaluating the properties in a centralized manner, and Section 3.3 describes the design of a more efficient and distributed approach.

3.1 How to Write Properties

The recommended way to write properties is to begin by thinking about the correct system behavior under steady-state operation. After identifying the desirable behavior, the programmer can write liveness properties to verify that such behavior is upheld during the program execution. If there are any liveness violations during the execution, the programmer can leverage insight from those violations to specify additional safety properties. These properties contain more specific checks to help narrow down the bugs that are causing the violations.

To write the actual properties, the programmer uses the grammar specified by the Mace compiler. This grammar was designed to be both concise and expressive. A simplified version of the grammar can be seen in Figure 3.2. Using this grammar, we were able to write safety and liveness properties for many of our Mace services (see Table 3.1). Most of the properties can be expressed concisely in just a few lines of code. This helps satisfy our requirement of making MaceODB easy to use. For example, consider property AllJoined in Table 3.1. This property is a liveness property, and its purpose is to check that all the nodes are eventually in the `joined` state. Notice how this property can be written succinctly in just one line of code. This simplicity is what makes MaceODB so easy to use. In this property, `nodes` is a keyword that represents all the nodes in the system, `state` is a default Mace variable that holds the current state of a particular node, and `joined` is one of the possible values for the `state` variable. By default, all Mace services have an implicit state value named `init`. The programmer can define more states, such as `joined` in this case, by listing

those values in the `state` block of his or her Mace service code. For more information on the concept of states in Mace, please refer to the Mace paper [5].

Let us now look at a more sophisticated property. Consider property Timer in Table 3.1. In this property, `recovery` is a timer object defined previously by the programmer inside the RandTree service code (see Section 5.1). It has a method called `nextScheduled()`. This method returns the next scheduled time. The purpose of this property is to check that for each node in the system, once it is done with the `init` state, the `recovery` timer should be scheduled. That way, when a failure occurs, there is a timer to trigger the recovery process.

The two examples above demonstrate the ease and simplicity of writing properties for MaceODB. The syntax is designed to be both concise and expressive. The programmer is only required to provide the bare minimum information of what he or she wants to check. Typically, this involves specifying the operations, and the data on which the operations are executed on. MaceODB and the Mace compiler will process the provided information, and generate all the low-level code to handle the actual property evaluation. All the details of how data is transmitted, collected, and evaluated are hidden from the programmer. This design helps achieve our goal of making MaceODB easy to use.

3.2 Centralized Property Evaluation

In the original design of MaceODB, we used a centralized approach for evaluating the properties. With this design, there is a central server that is responsible for evaluating all the properties across the entire system (see Figure 3.3). At a high level, this design consists of two main components: the Data Exporter module and the Property Checking module. The Data Exporter module operates on each node. Its task is to extract data that describes the current state of interests, and forward that data, together with a timestamp, to the central server. As for the Property Checking module, it is only

Table 3.1: Examples of properties that are used in some of the existing Mace services (Pastry, Chord, RandTree, MaceTransport)

Name	Property

LeftRight (Pastry)	*Test that size of leafset = sum of left and right set size.*
	`\forall n \in \nodes : {`
	`  n.myright.size() + n.myleft.size() = n.myleafset.size()`
	`};`

KeyMatch (Pastry)	*Test the consistency of the key of the node to the right.*
	`\forall n \in \nodes : {`
	`  n.getNextHop(n.range.second, −1).range.first =`
	`  n.range.second`
	`};`

AllNodes (Pastry)	*Test that all nodes are reached by following successor pointers from each node.*
	`\forall n \in \nodes : {`
	`  n.(getRight())* \eq \nodes`
	`};`

SuccPred (Chord)	*Test that a nodes predecessor is itself if and only if its successor is itself.*
	`\forall n \in \nodes : {`
	`  n.predecessor.getId() = n.me \implies`
	`  n.getSuccessor().getId() = n.me`
	`};`

PredNotNull (Chord)	*Test that predecessor pointer is eventually not null.*
	`\forall n \in \nodes :`
	`  \not n.predecessor.getId().isNullAddress();`

Timer (RandTree)	*Test that either the node state is init or recovery timer is scheduled.*
	`\forall n \in \nodes : {`
	`  (n.state = init) \or (n.recovery.nextScheduled() != 0)`
	`};`

OneRoot (RandTree)	*Test that there is exactly one one root.* `\for{=}{1} n \in \nodes : (n.root = n);`
AllJoined (RandTree)	*Test that eventually all the nodes will join the system.* `\forall n \in \nodes : n.state = joined;`
RetransTimer (MaceTransport)	*Test that retransmission timer is scheduled.* `\forall n \in \nodes : n.checkRetransmissionTimer();`

Property –> GrandBExpression

GrandBExpression –> (BExpression Join) BExpression

JoinExpression –> or | and | xor | implies | iff

BExpression –> Equation | BinaryBExpression | Quantification

Equation –> NonBExpression Equality NonBExpression

BinaryBExpression –> ElementSetExpression | SetSetExpression

Equality –> == | != | >= | <= | > | <

NonBExpression –> Variable NonBExpressionOp Variable

NonBExpressionOp –> + | −

ElementSetExpression –> Variable SetOp Variable

SetOp –> in | not_in

SetSetExpression –> Variable SetComparisons Variable

SetComparisons –> subset | propertysubset | eq

Quantification –> Quantifier Id Variable : GrandBExpression

Quantifier –> forall | exists | for{Number}

Figure 3.2: Simplified grammar for writing properties

used by the central server. This module consists of different methods that contain instructions on how to evaluate the properties. Upon receiving all the necessary data from the Data Exporter module, the central server invokes the Property Checking module to perform the property evaluation, and to report back to the programmer if there are any property violations.

To see how the Data Exporter and Property Checking modules work in action, let us consider property LeftRight in Table 3.1. With this property, we want to compare the size of `myleafset` versus the size of `myleft` plus the size of `myright`. Obviously, the data of interest are `myleafset.size()`, `myleft.size()`, and `myright.size()`. The Data Exporter module extracts this data from each node, and exports it to the central server. When the central server receives this data, it invokes the Property Checking module to perform the actual evaluation. This process consists of making method calls to iterate through the data from each node, and to evaluate whether or not `myleafset.size()` is equal to `myleft.size()` + `myright.size()`. If they are not equal, the evaluation process returns false, indicating that there is a property

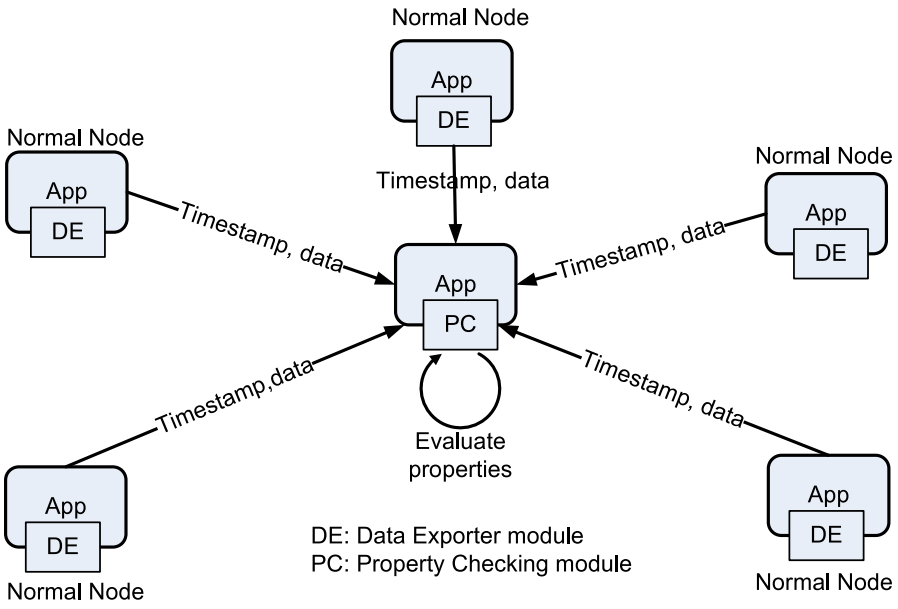


Figure 3.3: Centralized approach for evaluating properties

violation.

Our performance evaluation shows that the centralized approach is sufficient for most of the properties. However, certain properties require each node to send huge amounts of data to the central server. The central server then becomes a bottleneck, slowing down the whole property evaluation process, and reduces the overall performance of the system. We try to optimize our design by reducing the amount of data being sent. Instead of blindly sending all the data, we use a binary diff tool [14] to compare the differences between the old snapshot and the current snapshot of the data exposed by the Data Exporter module, and only send the diff data. This optimization significantly reduces the bandwidth being used. However, the central server is still the bottleneck, since it now uses a significant amount of memory and CPU to process the diff and to recreate the snapshots of the state data. Because of this poor performance, we decided to move toward a more decentralized approach.

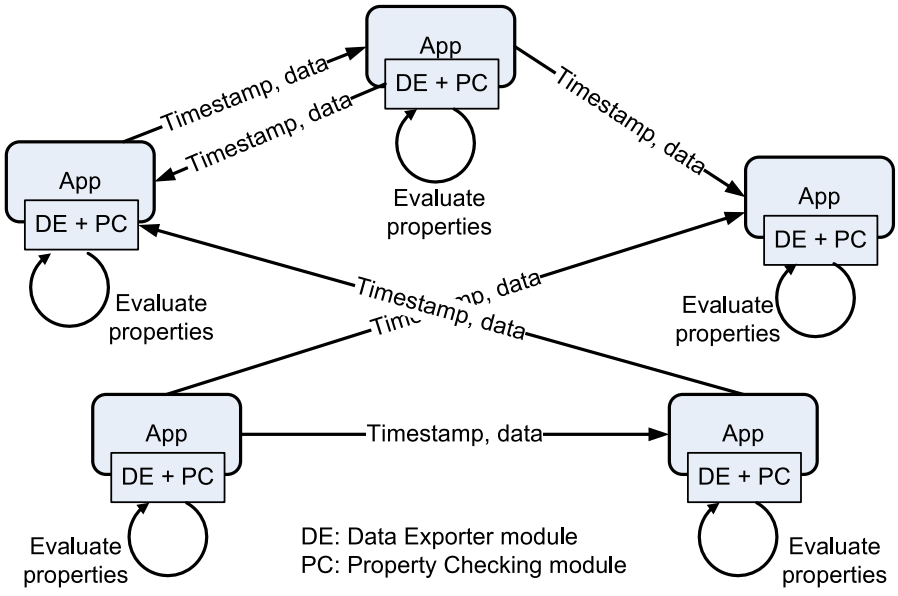


Figure 3.4: Decentralized approach for evaluating properties

3.3 Decentralized Property Evaluation

In the decentralized approach, MaceODB also generates code for the Data Exporter and Property Checking modules. The basic functionalities of these modules still stay the same. However, unlike the centralized approach, the Property Checking module is no longer used by a single central server, but instead is used in all of the nodes (see Figure 3.4). This modification is necessary because each node is now responsible for evaluating the properties by itself. This design helps eliminate the problem of having the central server as the source of bottleneck. It also makes the tool more fault-tolerant by removing the single point of failure. Additionally, a membership service is added to handle cases of network and node failures, which further satisfies the fault-tolerance requirement.

With the decentralized approach, the properties are now represented as dataflow graphs. Figure 3.5 shows an example of such representation for the LeftRight property. In general, with dataflow graphs, there are three main components: the leaves, the vertices and the arcs. The leaves correspond to the data that is used by the properties. This data can come from the same node on which the properties are being evaluated, or

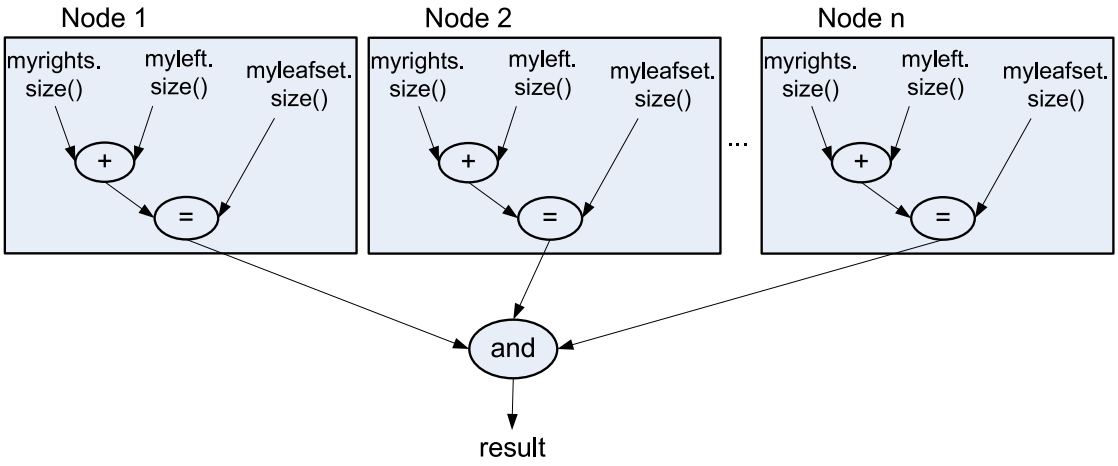


Figure 3.5: Example of using dataflow graph to represent the LeftRight property

it can come from other nodes. In the second case, the data would be made available via the Data Exporter module. As for the vertices, they represent the operations that must be performed in order to evaluate the properties. Together, these operations form the basic block for the Property Checking module. Last but not least, the arcs represent the input/output flows. They describe the dependencies between the operations.

At runtime, instances of these graphs are created for each timestamp. The vertices of the graphs can be evaluated as soon as the upstream inputs are available. The Property Checking module processes each vertex and evaluates the ones that are ready. Using this approach, vertices of different timestamps can be evaluated simultaneously in a pipelined fashion, thus exploiting the parallelism of the properties. This demonstrates the benefit of representing properties as dataflow graphs. Using this representation, we can abstract the data, the operations and the input/output dependencies, which in turn, makes it easier to decouple the properties into independent blocks that can be evaluated simultaneously in a distributed and parallel manner.

It is important to note that both the Data Exporter and the Property Checking modules are automatically generated by MaceODB. This is true for the centralized and decentralized approach. The programmers do not have to write any additional code. All they have to do is write the properties, and MaceODB will generate all the low-level code for exporting state data and evaluating the properties. This design helps satisfy our

requirement of making MaceODB easy to use. For more information on how MaceODB generates the code for the Data Exporter and Property Checking modules, please refer to the implementation details described in Chapter 4.

Chapter 4

Implementation

This chapter presents the implementation of MaceODB. For completeness, both of the implementations for the centralized and decentralized approach are presented. The former is covered in Section 4.1, and the latter is covered in Section 4.2. The rest of the chapter covers implementation details that apply to both.

4.1 Centralized Implementation

This section describes the implementation for the centralized approach. In particular, it shows the steps that MaceODB must go through in order to construct the Data Exporter and Property Checking modules.

4.1.1 Data Exporter Module

As mentioned in Section 3.2, this module is responsible for exporting data to the central server. The exported data is eventually used by the central server in order to evaluate the properties. Therefore, an important task in building the Data Exporter module is to determine what data it needs to send. In the simplest implementation, MaceODB uses the grammar specification in Figure 3.2 to parse through each property, and identifies all the variables that are used in that property. The values of these variables correspond to the data that need to be sent to the central server. This simple-minded

implementation works for most of the properties. However, there are cases where this implementation is inefficient, and also cases where it fails to work.

First, let us look at how this implementation can be inefficient. Consider property Timer in Table 3.1. Using the simple-minded implementation, MaceODB identifies the following variables as data that need to be exported: `n.state`, `init`, `n.recovery.nextSchedule()`, and `0`. It is, however, inefficient to export all of those variables. As it turns out, the Timer property can actually be evaluated locally within a node without the need of exporting any data. This optimization is possible because the data required for the property are either constants or are variables that come from the same node. Therefore, instead of sending all the data to the central server, each node can evaluate the property by itself, and only forward the result from the evaluation process. This optimization helps reduce the amount of data being sent to the central server, which in turn reduces the chance of overflowing the server's bandwidth.

Now, let us look at cases where the simple-minded implementation fails to work. These cases involve properties that include methods whose parameters require data inputs from other nodes. With this type of property, MaceODB wants each node to export the results from executing the method calls. Unfortunately, the nodes are not capable of making the method calls by themselves. This limitation is there because the methods require data inputs from other nodes. To solve this problem, we need to have the central server imitate the method calls for each of the nodes. In order to do that, each node has to send its entire state dump to the central server. These state dumps allow the central server to create dummy objects, which imitate the nodes from which the state dumps come from. Using these dummy objects, the central server can imitate the method calls using the appropriate parameter inputs, and then proceed with the property evaluation process.

After figuring out what data to send, the next phase in building the Data Exporter module is to generate the actual code for sending the data. Luckily for us, this is an easy task since Mace already provides a mechanism for doing that. Specifically, Mace provides a C++ class called `Message`. This class allows programmers to write

code for sending messages from one node to another. MaceODB takes advantage of this class, and uses it as the mechanism for exporting data to the central server.

4.1.2 Property Checking Module

The next step is to implement the Property Checking module. As mentioned in Section 3.2, this module is used by the central server. Its job is to evaluate the properties and to alert the programmer of any property violations. The first step in constructing this module is to determine how each property can be evaluated. MaceODB accomplishes this task by parsing through the properties, and breaking them into smaller expressions as specified by the grammar in Figure 3.2. These expressions correspond directly to the operations that must be performed in order to evaluate the properties. For example, consider the property Timer in Table 3.1. When MaceODB parses that property, it identifies the following expressions/operations:

- Equation Expression #1 - This expression compares `n.state` and `init`.
- Equation Expression #2 - This expression compares `n.recovery.nextScheduled()` and `0`.
- Join Expression - This expression performs a logical OR operation on the results of the previous 2 expressions.
- Quantification Expression - This expression performs a `forall` loop operation.

After identifying the above expressions, MaceODB proceeds to the next phase of generating the actual code for the Property Checking module. For each of the expressions that it sees, MaceODB generates a C++ method. This method contains code for the operation that the expression is associated with. Together, these methods form the basic block for the Property Checking module.

4.2 Decentralized Implementation

This section describes the implementation for the decentralized approach. For most parts, this implementation is similar to the previous one. The main differences are in the ways the Data Exporter and Property Checking modules are constructed. The following subsections present the implementations in more details.

4.2.1 Data Exporter Module

For the decentralized implementation, the Data Exporter module is slightly different from its centralized counterpart. Instead of exporting data to the central server, this module is now used by each node to exchange data among each other. To help this module carry out that task, MaceODB generates two classes: `RequestMessage` and `ReplyMessage`. These classes extend the `Message` class, which is defined by the Mace C++ extension library. As their names imply, the `RequestMessage` class is used for requesting data, and the `ReplyMessage` class is used for returning the result. These classes provide the basic mechanism for exchanging data between nodes.

To see how this implementation works in action, consider the property KeyMatch in Table 3.1. This property has an equality operation that compares `range.first` and `range.second`. The `range.second` input is from the node on which the operation is being executed. As for the `range.first` input, it actually comes from the node that is specified by the result of `n.getNextHop(n.range.second, -1)`. Until that input is available, the equality operation can not be executed. To handle this situation, the executing node uses the Data Exporter module to request the needed input. It sends a `RequestMessage` to node X, where X is the return value from calling `getNextHop(n.range.second, -1)`. When node X receives the request message, it replies with the `ReplyMessage` object, which contains `range.first`, the requested data. Upon receiving `ReplyMessage`, the original node extracts the returned data, and pipes it into the input field for equality operation. The equality opera-

tion is now ready to be executed.

4.2.2 Property Checking Module

As mentioned in Section 3.3, the Property Checking module consists of operations that provide instructions on how to evaluate the properties. In terms of dataflow graph, these operations correspond to the vertices of the graphs. To generate code for the vertices, MaceODB parses through the properties and identifies all the operations that are parts of the property checks. Then, for each operation that it finds, MaceODB generates a C++ class to represent that operation. This step is different from the centralized approach, where we generate methods instead of classes. The reason for using classes is because we want a more flexible and object-oriented way for constructing the dataflow graphs. With classes, we can set up the vertices as objects whose member variables contain data inputs for the operations, and whose methods are the operations themselves. Specifically, the objects will contain the following methods:

- isReady() - returns true if all the needed inputs are available and if the operation is ready to be executed. Otherwise, returns false. In terms of the dataflow graph, this corresponds to whether or not the upstream inputs are available for execution.
- eval() - executes the operation. When the execution is complete, the output is piped into other operations that depend on it. In terms of the dataflow graph, this corresponds to evaluating the operation represented by the corresponding vertex, and then passing the result to the downstream vertices.
- getStatus() - returns 0 if the operation has not been executed, returns 1 if the operation is in the process of being evaluated, and returns 2 if the operation has finished.

At runtime, instances of these classes are created and stored inside a queue. Then in a separate method, we iterate through the queue, and evaluate any operations that are ready for execution. In terms of the dataflow graph, this corresponds to the process of traversing the graph and evaluating any vertices that are ready. This process is done on a per-node basis, thus allowing the properties to be evaluated in a distributed and parallel

manner.

4.2.3 Closure

An interesting scenario is when the properties include operations that involve closure sets. For example, consider property AllNodes in Table 3.1. In this property, the closure set is denoted by the (*) symbol. The basic idea here is that for each node n, the closure set of `n.getRight()` should be equal to all of the nodes in the system. In other words, if we start at node n, and traverse to the right until we run out of nodes, or until we come back to a previously visited node, then at the end, we should have visited all of the nodes within the system. To support this type of operation, MaceODB instructs the Mace compiler to generate a `ClosureMessage` class. This class contains the following member variables:

- originNode - holds the value of the first node in the closure set.
- closureSet - holds the list of nodes that have been visited so far. At the end, this list corresponds to the result for the closure set.

MaceODB also defines a transition method which determines how the closure set can be constructed. Figure 4.1 shows the pseudocode for that method.

```
Input: ClosureMessage (originNode, closureSet)
  if myself == ClosureMessage.originNode then
     result = ClosureMessage.closureSet
     finish
  else if myself is in ClosureMessage.closureSet then
      forward ClosureMessage to ClosureMessage.originNode
  else
      ClosureMessage.closureSet.insert(myself)
      forward ClosureMessage to next node
  endif
```

Figure 4.1: Pseudocode for calculating closure set

4.3 Guessing Input/Output Type

When writing properties, the programmer does not have to worry about specifying the types of the variables. The benefit of this approach is that it makes the task of writing properties much easier. The downside is that MaceODB will have to do extra work in order to determine the types for those variables. Fortunately, MaceODB can use the existing functionality of the Mace compiler to assist in identifying the types. This works for the majority of the time, but not always. When the Mace compiler fails to identify the type, MaceODB will attempt to guess the type in a heuristic manner. One of the heuristics is to guess the type based on the context in which the variable is being used. For example, consider property LeftRight in Table 3.1. For this property, the Mace compiler is only able to identify the types for `myright`, `myleft` and `myleafset`, but it is not able to determine the return type for the `size()` method. To handle this, MaceODB looks at how the variables are being used. In this case, `n.myright.size()` and `n.myleft.size()` are used in an addition operation. Therefore, it is safe to guess that `n.myright.size()` and `n.myleft.size()` are of type numeric. As for `n.myleafset.size()`, it is used in an equality operation, which has a left-hand-side operand of type numeric. Therefore, `n.myleafset.size()` is of type numeric as well.

4.4 Globally Consistent Snapshot

Many of the properties are required to be evaluated across all the nodes. In order to evaluate these properties, we need to have a consistent snapshot across the entire system. To support this, we added a logical clock [12] to the Mace language. Each node in the system maintains its own logical clock, which starts at 0, and increases every time there is an event transition. Each time a node sends a message, it attaches its logical clock to the message. Upon receiving the message, the receiving node updates its logical clock to be the maximum of its local logical clock and the clock attached in the message. This helps establish the happens-before relationship, and allows the logical

clock to be used as a global timestamp. Using this mechanism, MaceODB can associate each node's data with a timestamp, allowing it to later on sort through the data and order them into consistent snapshots.

4.5 Membership Service

Before MaceODB can evaluate the final result for a particular timestamp, it needs to have a complete snapshot for that time. A snapshot for timestamp t is considered to be complete if it contains data from all the nodes that participate in the system at time t. This means MaceODB needs to know the nodes membership at any given time. To handle this, MaceODB sets up a membership server. This server is responsible for maintaining the membership list across the whole system. Whenever a node joins the system, it registers itself by sending a message to the membership server. The membership server then forwards the update to all the other participating nodes, letting them know about the new membership addition. To handle node failures, the membership server uses a heartbeat mechanism to periodically query the participating nodes. Upon receiving this query, each node sends back a reply message, letting the membership server know that it is still up and running. If the membership server does not receive a reply message from a particular node for a certain amount of time (specified by the user), that node is considered to be down. The membership server then sends an update message to other nodes, telling them to remove the failed node from their membership lists.

4.6 Evaluating Liveness Properties

By definition, a liveness property is not always required to be true. It only asserts that something good will eventually happens. Under this definition, a violation of liveness property does not mean there are problems with the code. It simply means that the system is undergoing changes that might lead to undesirable states. Therefore, in order to correctly evaluate liveness, we must wait for the entire program execution to be

complete. This requirement makes liveness property not practical for online debugging, since sometimes, distributed systems are meant to be left running indefinitely. To solve this problem, we modify our definition of liveness property to be bounded by a time window. Under this new definition, a liveness property asserts that something good should eventually happen within x amount of time, where x is a logical time value that can be specified at runtime by the programmer. During the evaluation process, if the property evaluates to false, MaceODB will check the elapsed time to determine how long the property has been violated. If that time is greater than the specified window, then MaceODB will report the violation as an error. Otherwise, the violation is ignored.

Chapter 5

Experience

We have used MaceODB to test a variety of systems implemented in Mace, including MaceTransport, RandTree, Pastry [16], Chord [18], Scribe [17], SplitStream [2] and Paxos [10]. Most of these systems are mature, stable, and have been tested extensively in the past. Therefore, we were not able to find many existing bugs. However, we were able to find non-trivial bugs in RandTree, Chord and Paxos.

5.1 RandTree

RandTree implements a random overlay tree that is resilient to node failures and network partitions. It serves as a backbone for a number of high level applications such as Bullet [9] and RanSub [8]. An important liveness property that RandTree must hold is that there should be only one overlay tree. In case of network and node failures, this property will not hold. To address this issue, a recovery timer was added. This timer periodically checks to see if there was a network partition, and invokes the recovery process if needed. Using property Timer in Table 3.1, we were able to find a bug where the recovery timer was not scheduled correctly. When running RandTree with MaceODB enabled, the property Timer evaluated to false, which indicated that for some nodes, neither the state was "init" nor the recovery timer was set. Using that knowledge, we went back to the source code and checked where the recovery timer was scheduled. Figure 5.1 shows an excerpt of the code that contains the bug.

```
joinOverlay(const NodeSet& peerSet, registration_uid_t rid) {
    if (peers.empty()) {
        state = joined;
        ...
    }
    else {
        state = joining;
        ...
        recovery.reschedule(TIMEOUT);
    }
}
```

Figure 5.1: RandTree bug

The problem with the code is that `recovery.reschedule(TIMEOUT)` never get called if `peers` is empty. To fix the bug, we need to move that statement out of the else block. That way, the recovery timer is always scheduled whenever a node joins the overlay network. An interesting note is that this same property was used previously in the Mace model checker, and yet, the model checker failed to catch the bug. This failure is caused by the way the system was set up for the model checking. The programmers set it up in such a way that whenever a node joins the system, it always joins together with another peer. So when the code above is executed, `peers.empty()` will always return false, causing the execution flow to go to the else block, which then causes the recovery timer to be scheduled. The property then always evaluated to be true. This example is a perfect demonstration of the disadvantages of using model checker: the checking is done in a specialized environment, which can be quite different from how the real system would behave. This clearly demonstrates the value of having a tool such as MaceODB, that allows the checks to be done in real time and on real, live systems.

5.2 Chord

Chord is a P2P distributed lookup protocol [18]. It supports a single operation of mapping keys to nodes. Using Chord, the nodes are set up in a ring topology. Each node in this ring has pointers to its successor and predecessor. To join the ring, a new node gets its predecessor and successor from another node. Then, to insert itself into the ring, it tells the successor node to update its predecessor pointers. Finally, there is a stabilize process, which helps ensure that global successor and predecessor pointers are consistent and correct.

The predecessor pointers are used in the lookup and stabilize process. It is important that they are updated correctly, especially in the presence of node churns and node failures. To test our implementation of Chord, we use a liveness property (see Table 3.1) to check that eventually, all the predecessor pointers are not null. This minimal check helps ensure that in case of node failures, the predecessor pointers will eventually point to other valid nodes. Using this property, we set up an experiment where we simulate node failures. We observed that in a system of n nodes, if n-1 nodes go down, the predecessor pointer of the remaining node will become null. It will eventually fix itself if the failed nodes come back up. However, if they remain down, the predecessor pointer will remain null for the rest of the program execution. The correct behavior is that the remaining node should have updated its predecessor pointer to be itself. Fortunately, this bug is quite trivial due to the rarity under which there is a failure of n-1 nodes. Nevertheless, this is a good demonstration of the effectiveness of MaceODB in finding rare bugs that would have been otherwise overlooked.

5.3 Paxos

Paxos is an efficient and highly fault-tolerant algorithm for reaching consensus in a distributed system. The goal behind this algorithm is for a collection of processes to agree upon a value. With Paxos, the processes are run on nodes that belong to the following roles: proposers that propose values, acceptors that cooperate to select a single

```
void proposalChosen(const MaceKey& src, const Proposal& p, log_index_t unanimousApplied) {
  ...
  freeValues(lk, p.view, chosen);
  freeValues(lk, p.view, pending);
  ...
}

void freeValues(const PaxosLogKey& lk, const View& view, ChosenValueMap& m) {
    ChosenValueMap::const_iterator i = m.lower_bound(Key::getMin(lk));
    while (i != chosen.end() && i->first.equalsLog(lk) &&
            (view.n == 0 || i->second.view.n < view.n)) {
      i = chosen.erase(i);
  }
} // freeValues
```

Figure 5.2: Paxos bug

proposed value, and learners that must figure out what value has been chosen. The algorithm has been discussed extensively elsewhere [3, 10, 11], so we will not go into details on how it works. Instead, we will present our Paxos implementation together with the bug that we found.

For our Paxos implementation, we have 3 nodes that are responsible for the consensus algorithm. One of them will be the leader. The other two will both be acceptors and learners. At any given time, the membership of these three nodes can be changed or replaced. Therefore, we have safety and liveness properties to verify that the membership stays correct. To carry out the test, we write a simple driver program that loops through a list of all the available nodes, and performs different membership proposals. For example, node A begins by proposing A, B and C. Node B then proposes B, C and D, and so on. While running this program, our implementation of Paxos crashes with a segmentation fault. Using gdb, we track down the code that is causing the problem, and present it in Figure 5.2. In this code snippet, the method `freeValues` is called whenever a node learns that it is no longer in the membership list. This method takes a map `m` by reference, does a `lower_bound` on it, and then compares that iterator

```
void proposalChosen(const MaceKey& src, const Proposal& p, log_index_t unanimousApplied) {
  ...
  freeValues(lk, p.view, chosen);
  freeValues(lk, p.view, pending);
  ...
}

void freeValues(const PaxosLogKey& lk, const View& view, ChosenValueMap& m) {
    ChosenValueMap::const_iterator i = m.lower_bound(Key::getMin(lk));
    while (i != m.end() && i->first.equalsLog(lk) &&
           (view.n == 0 || i->second.view.n < view.n)) {
      i = m.erase(i);
  }
} // freeValues
```

Figure 5.3: Fixes for the Paxos bug

(and erases elements) from the map `chosen`. The problem here is that `freeValues` is called with both `chosen` and `pending` as `m`, and when called with `pending`, the code is iterating and erasing elements from the wrong map. The corrected code can be seen in Figure 5.3.

Even though we stumble upon this bug via the segmentation fault, and not via the result of some failed property check, this example still demonstrates the usefullness of MaceODB. Without MaceODB, we would not bother to write the driver program to test the Paxos implementation in the first place. Therefore, one can argue that MaceODB has indirectly helped us discover the bug.

Chapter 6

Performance Evaluation

To evaluate the performance of MaceODB, we performed a macro-benchmark that measures the overhead of using MaceODB. We hoped to show that MaceODB is lightweight, and that it incurs a minimal amount of impact on the systems under test. Additionally, we performed a micro-benchmark that measures the time it takes to evaluate different types of properties. The purpose of this benchmark is to help programmers be aware of properties that are expensive to evaluate, and to help provide insights for future work.

6.1 Macro-benchmark

For the macro-benchmark, we measure the impact that MaceODB incurs on appmacedon. appmacedon is a data streaming application that can run on top of any multicast or unicast services. For our experiments, we run appmacedon on top of RandTree, Scribe and SplitStream. For each of these services, we run appmacedon with and without MaceODB, and compare the differences in goodput, memory usage and CPU usage.

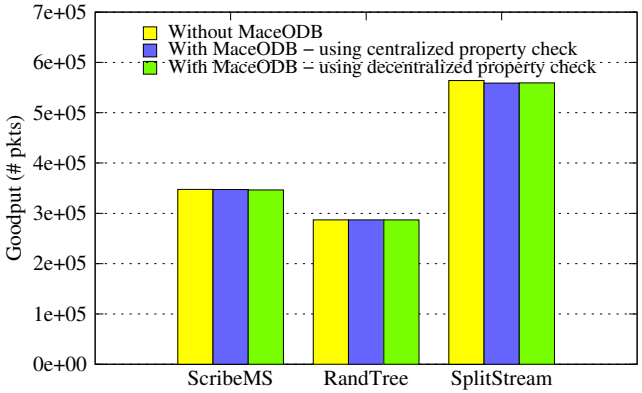
We run our experiments on several different setups that range from small systems of 25 clients to larger systems of 100 clients. These clients are emulated on 17 physical machines via the ModelNet network emulator [19]. Each of the physical machines is a dual Xeon 2.8 MHz processor with 2GB of RAM. The emulated topologies

consist of an INET network with 5000 nodes. The emulated clients have bandwidth of 6,000-10,000 Kbps, and latency of 2-40 ms.

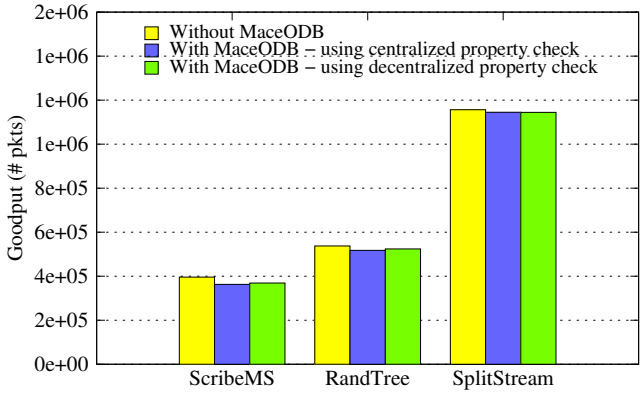
To deploy our experiments, we use Plush [1], a fully configurable application management infrastructure that provides a unified set of abstractions for specifying, deploying and monitoring distributed systems. We also use Mission, a batch scheduler for submitting our experiments to be run on ModelNet. To start our experiments, we instruct Plush to instantiate a single appmacedon instance that acts as the data source and also as the bootstrap node. Then, we tell Plush to start the other instances of appmacedon. These instances will use the first instance to bootstrap themselves into the system, and start streaming data once they are joined.

6.1.1 Goodput

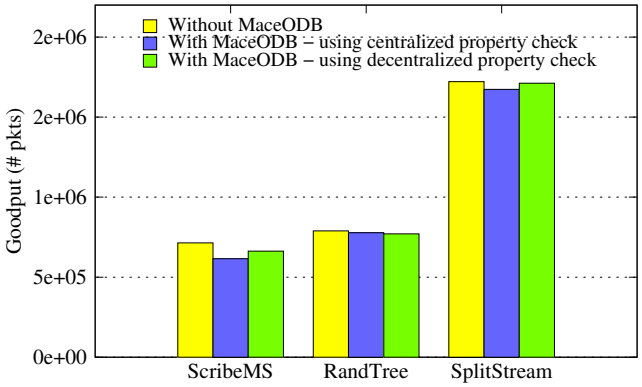
Each of our experiments is run for 5 minutes. At the end of each run, appmacedon reports the number of packets that were sent and received. Using this data, we construct different graphs to compare the goodput of appmacedon when running with and without MaceODB (see Figure 6.1). Additionally, we calculate the impact that MaceODB incurs on the system's goodput by taking the ratio of the loss in goodput over the original value when the system was run without MaceODB. We present the results in Table 6.1. According to the results, MaceODB performs quite well for systems with 50 nodes or less. For these systems, the impact for most cases is less than 7%. As for larger systems, the results are quite different. With the centralized approach, the performance is very poor when running on Scribe and SplitStream. The impact on the goodput is as high as 13.8% for systems of 75 nodes and up to 20.17% for systems of 100 nodes. This poor performance is not a surprise for us, since we know the centralized approach is not scalable. As for the decentralized approach, the performance is much better. Its impact on the goodput is only around 7% or less. More importantly, the results also indicate that as we increase the size of the system, the impact on performance increases very slowly. This trend allows us to believe that the decentralized approach of MaceODB is scalable for large systems.



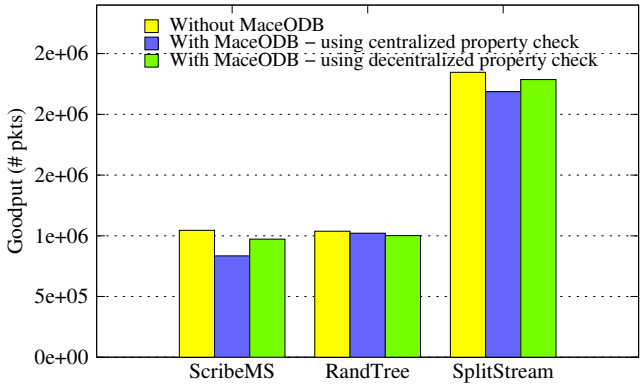
(a) Running on 25 nodes



(b) Running on 50 nodes



(c) Running on 75 nodes



(d) Running on 100 nodes

Figure 6.1: Goodput of appmacedon when running on top of different services with and without MaceODB.

Table 6.1: Impact of using MaceODB

		Impact on Goodput	
Services	Number of Nodes	Centralized Approach	Decentralized Approach
RandTree	25 nodes	0.01%	0.05%
	50 nodes	1.68%	2.53%
	75 nodes	1.40%	2.29%
	100 nodes	1.58%	3.53%
ScribeMS	25 nodes	0.07%	0.33%
	50 nodes	8.35%	6.77%
	75 nodes	13.83%	7.16%
	100 nodes	20.17%	7.01%
SplitStream	25 nodes	0.93%	0.84%
	50 nodes	1.01%	1.07%
	75 nodes	2.19%	1.57%
	100 nodes	6.78%	2.55%

To understand why the centralized approach performs so poorly for large systems, we identify the different sources of overhead in using MaceODB, and compare how the overhead from each of those sources differ between the centralized and decentralized approach. The first source of overhead is from the additional data being sent via the Data Exporter module. To calculate this type of overhead, we first run our experiments on 100 nodes without MaceODB enabled and measure the number of bytes that are sent and received. Then, we run our experiements again, but with MaceODB enabled for each of the properties, and measure the additional number of bytes that are introduced into the system. We then take the ratio of that value over the original number of bytes that were sent and received when running without MaceODB. We define this ratio as the data overhead introduced by the Data Exporter module and present the results in Table 6.2. According to the results, the centralized approach actually exports less data than the decentralized approach. Therefore, this type of overhead cannot be the reason why the centralized approach performs so poorly.

Let us look at the second source of overhead, which is the computational cost for evaluating the properties. As later shown in Section 6.1.2 and 6.1.3, the centralized approach uses a significant amount of memory and CPU. With this approach, each node

Table 6.2: Data overhead introduced by the Data Exporter module. For more information regarding the properties, please refer to Table 3.1.

	Data Overhead	
Property	Centralized Approach	Decentralized Approach
LeftRight	0.03%	0.06%
KeyMatch	0.05%	0.09%
AllNodes	0.05%	8.79%
SuccPred	0.08%	0.21%
PredNotNull	0.05%	0.10%
Timer	0.03%	0.05%
OneRoot	0.02%	0.05%

is responsible for computing the binary diff, which is a CPU and memory intensive operation. As for the central server, it is responsible for evaluating the properties for the entire system. For large systems of 100 or more nodes, the central server cannot process the properties fast enough. It becomes the bottleneck, slowing down the entire system. This explains why the centralized approach performs so poorly. As for the decentralized implementation, the computational cost is divided among the nodes, thus making this approach more efficient and scalable.

6.1.2 Memory Usage

Besides measuring the goodput, we also measure memory and CPU usage. Figure 6.2 shows the results of memory usage during the 5-minute run. Without MaceODB, appmacedon uses approximately 18 MB of RAM. Using this as the base value, we compare it against the memory usage of appmacedon when it is run with MaceODB enabled. Our results show that depending on how we evaluate the properties, the impact on memory usage is quite different. With the decentralized approach, the memory usage is around 20 MB, which is slightly higher than the base value, but is still acceptable. As for the centralized approach, the memory usage on the central server is significantly high. Even with a cleanup mechanism, the central server still uses as much as 200 MB of memory. The reason for such high memory usage is because the central server is responsible for storing all the data that are forwarded from the other

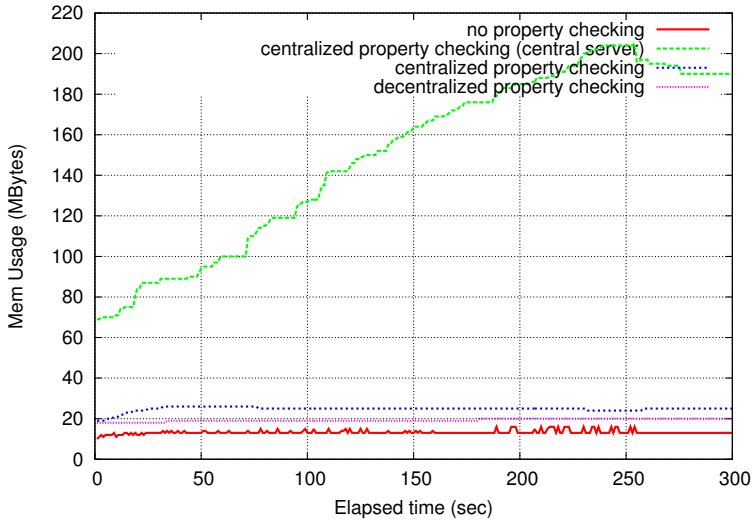


Figure 6.2: Memory usage of appmacedon when running with and without MaceODB

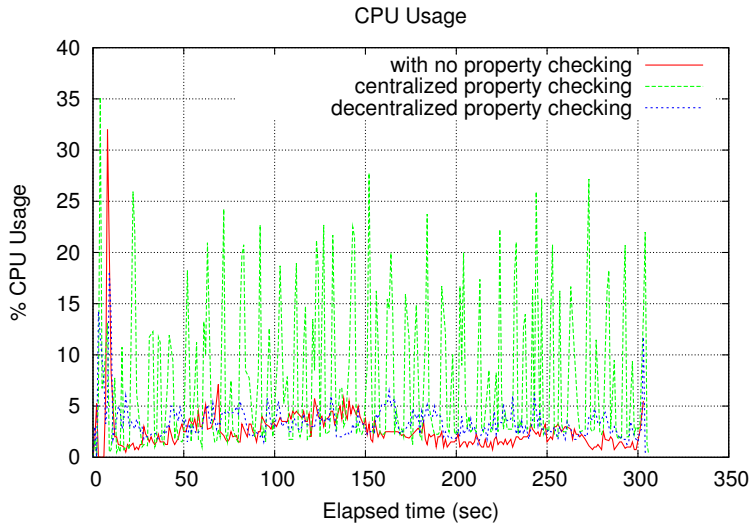


Figure 6.3: CPU usage of appmacedon when running with and without MaceODB

nodes. These data are stored in memory and cannot be deleted until they form a complete snapshot of the whole system. At that point, the central server can proceed to evaluate the properties using the newly created snapshot. Once that is done, the data can then be removed.

6.1.3 CPU Usage

Figure 6.3 shows the CPU usage during the same 5-minute run. Without MaceODB, the CPU usage fluctuates around 2-3%. With MaceODB, the CPU usage is only slightly higher if we use the decentralized approach for evaluating the properties. On the other hand, with the centralized approach, the property evaluation process is quite CPU-intensive. During the 5-minute run, there were spikes in the CPU usage that are as high as 28%. This further confirms the fact that the centralized approach is not scalable.

Overall, the macro-benchmark provides us with two important implications. First, the centralized approach is not scalable. When running on large systems, this approach causes a noticeable drop in goodput, and significant overhead in memory and CPU usage. Second, the decentralized approach is scalable and efficient for systems of 100 nodes. When using this approach, the impact on goodput is only 7% or less, and the memory/CPU overhead is quite low. We expect similar performance for larger systems. In conclusion, the decentralized approach allows us to satisfy our requirement of making MaceODB lightweight, thus allowing it to be left running on deployed systems without incurring too much impact on the system's performance.

6.2 Micro-benchmark

For the micro-benchmark, we want to measure the time it takes to evaluate different types of properties. The goal is to identify properties that are expensive to evaluate, and hopefully, that will provide us with insights on what needs to be improved. Similar to the macro-benchmark, we use Plush and Mission to run our experiments on ModelNet. This time, we run 100 instances of appmacedon on top of RandTree, Pastry and Chord. We modify the program to log the time before and after each property evaluation. The experiments are then run for 5 minutes. At the end of each run, we process the logs, and calculate the average time for each property evaluation. The result is shown in Table 6.3. From the result, one can see that the cost of evaluation varies

greatly among the properties. Some of them can be evaluated in as little time as 60 microseconds, while others require up to 9 seconds. To understand the reason behind this variance, we categorize the properties into different property types, and try to find a correlation between the type and the speed at which the property can be evaluated. Table 6.4 summarizes the findings. Basically, the least expensive properties are the ones that can be evaluated locally. These properties contain operations that do not require any data inputs from other nodes. Examples of such properties are LeftRight, Timer and OneRoot. With this type of properties, the cost of evaluation is a function of how fast the CPU can process each operation. For our particular setup, this value is approximately tens of microseconds.

Let us now look at properties that are more expensive to evaluate. Consider the following properties: KeyMatch, SuccPred, and PredNotNull. What these properties share in common is that they all contain operations whose inputs are data from other nodes. For these properties, the cost of evaluation is a function of how fast the operations can be executed and how fast the data can be transferred. Typically, the latter is the dominant factor. In other words, the speed of evaluating these properties depends directly on the bandwidth and latency of the network on which the system is deployed. For our particular network topology, the time it takes to send a ping message from one node to another is approximately 80 - 100 ms. The round trip time for sending and receiving a message is then 160 - 200 ms. This value matches up with our benchmark results, which show the cost of evaluating these properties at $>$ 200 ms.

The last group of properties is the one that involves closure set. This type of property is the most expensive to evaluate. For these properties, a significant amount of time is spent in calculating the closure set. The high computational cost is due to our inefficient algorithm. As described in Section 4.2.3, we calculate the closure set by forwarding messages from one node to another, until we come back to a visited node, or until we run out of node to visit. In the worst case scenario, we would have to visit all of the nodes that are in the system. When that happens, the cost of calculating the closure set is O(n$\times$t), where n is the number of nodes, and t is the time it takes to send a

Table 6.3: Amount of time it takes to evaluate different properties. For more information regarding these properties, refer to Table 3.1.

Property	Service	Evaluation Duration
LeftRight	Pastry	79 μs
KeyMatch	Pastry	210 ms
AllNodes	Pastry	9.1 secs
SuccPred	Chord	220 ms
PredNotNull	Chord	205 ms
Timer	RandTree	60 μs
OneRoot	RandTree	61 μs

Table 6.4: Time cost for evaluating different types of properties

Property Type	Time cost
Properties that can be evaluated locally within a node	$\sim$70 μs
Properties that require data from multiple nodes	$\sim$200 ms
Properties that involve closure set	$\sim$9 secs

message from one node to another. For our particular experiments, we have 100 nodes in the system, and the ping time is approximately 80 - 100 ms. Therefore, the cost of calculating the closure set is around 8 seconds to 10 seconds. This matches up with the result for property AllNodes, which requires 9.1 seconds to be evaluated. Obviously, the cost of evaluating this type of properties is quite high. We hope address this issue in our future work (see Chapter 8).

Chapter 7

Related Work

There are several related techniques for debugging distributed system. We will present them in this chapter, and compare the advantages and disadvantages of using those techniques with respect to MaceODB.

Model Checking. Several papers have proposed model checking as the mechanism for debugging distributed systems. With model checking, the programmer defines specifications for the system, and uses the model checker to systematically explore the state-space of the system, while checking if the specifications hold or not. This mechanism can be a powerful debugging tool, since extremely large state-spaces can often be traversed in just minutes, allowing the programmer to discover many bugs that would otherwise be very hard to find manually. However, the problem with using model checkers is that the checking process is typically done in a controlled and virtualized environment. This type of setup does not reflect the real environment on which the system will be deployed. This drawback, however, is not a problem for MaceODB, since it can run on live, deployed system. As a result, MaceODB does a better job at detecting realistic bugs that might only show up in the real environment, such as bugs that are caused by network and node failures.

Replay-based Checking. Much research has gone into replay-based checking. With this approach, the programmer has the ability to replay the program in the same order and environment as in the original run. A notable example of replay-based

checking is *liblog* [4]. This is one of the first replay tools to address large distributed system. It works by logging the execution of deployed application processes, and allows programmers to replay them deterministically. The benefit of using *liblog* or any other replay tool, is the ability to consistently reproduce bugs that occurred previously at runtime. This ability allows programmers to perform careful offline analysis. The weakness of replay-based checking is the high cost of logging and replaying the whole program execution, especially for large systems. An interesting point to note is that MaceODB and replay tools can be complementary to each other. A programmer can use MaceODB to detect bugs at runtime. Then using replay-based checking, the programmer can perform offline analysis to further debug the problems.

Log Analysis. Many systems focus on parsing through logs to perform post-mortem analysis. A notable example of this methodology is Pip [15]. With Pip, programmers specify expectations about a system's structure, timing and other properties. At runtime, Pip logs the actual behavior. Then, once the logs are collected, Pip provides the programmers with queries and a visual interface for exploring the expected and unexpected behavior. The main problem with Pip, and many other log-based analysis tool, is the high overhead in logging the data.

Other On-line Debuggers. MaceODB is very similar to D^3S [13]. Both share the idea of using predicates as the language for debugging distributed systems. They also share the idea of representing the predicates in term of dataflow graphs. MaceODB differs from D^3S in the way predicates are written. With D^3S, predicates are written in mixture of C++ and scripting language. When writing the predicates, the programmer has to specify the stages and the input/output dependencies. In term of dataflow graph, this means the programmer has to specify the vertices in the graph, and also specify how each vertex connects to each other. As for MaceODB, all the programmer has to do is write the predicates as properties in the Mace language and nothing else. The Mace compiler will generate all the necessary C++ code to represent the stages (vertices) and the connections among them (arcs). As the result, MaceODB is much easier and simpler to use. However, the disadvantage of MaceODB is that in some cases, it might be less

efficient than D^3S. With D^3S, the programmer can specify exactly what input and output data are required, while as for MaceODB, it might be generating suboptimal code that sends more data than what is necessary.

Chapter 8

Conclusions and Future Work

Debugging distributed systems is a challenging task. In this dissertation, we have presented MaceODB, a tool to help make that task easier. MaceODB provides programmers with the ability to perform online property checking for services written in Mace. It is easy to use, yet flexible and powerful enough to catch several non-trivial bugs in the existing Mace services. It is also fault-tolerant, and low in overhead, which makes it possible to be left running on live systems without incurring too much impact on the performance.

For future work, we hope to further improve the tool's performance. In particular, we want to improve the way we calculate closure sets. Currently, each node is responsible for doing the calculation by itself. This implementation is inefficient because it often results in the same operations being done on multiple nodes. To avoid this, a possible solution is to have designated nodes that are in charge of doing the calculation. With this approach, we can further improve the performance by applying dynamic programming methodology to our calculation process.

Another improvement that we want to make is to extend MaceODB to support more than just property checking. Using properties, a programmer can only write predicates that will return either true or false, but nothing else. This limited functionality is not helpful in situations where the programmer is interested in querying for some particular state data from the system. In order to support these scenarios, we would have to

extend MaceODB to support distributed queries. These queries would be very similar to our notion of properties, but instead of returning true or false, they will return some data specified by the programmer.

Bibliography

[1] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: Distributed application configuration, management, and visualization with plush. In *LISA'07: Proceedings of the 21st conference on 21st Large Installation System Administration Conference*, pages 1–19, Berkeley, CA, USA, 2007. USENIX Association.

[2] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313, New York, NY, USA, 2003. ACM.

[3] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, New York, NY, USA, 2007. ACM.

[4] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*. USENIX, 2007.

[5] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188, New York, NY, USA, 2007. ACM.

[6] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code (awarded best paper). In *NSDI*. USENIX, 2007.

[7] E. Kindler. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

[8] D. Kostić, A. Rodriguez, J. Albrecht, A. Bhirud, and A. Vahdat. Using random subsets to build scalable network services. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 19–19, Berkeley, CA, USA, 2003. USENIX Association.

[9] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: high bandwidth data dissemination using an overlay mesh. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 282–297, New York, NY, USA, 2003. ACM.

[10] Lamport. The part-time parliament. *ACMTCS: ACM Transactions on Computer Systems*, 16, 1998.

[11] Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32, 2001.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[13] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. In *NSDI*, 2008.

[14] C. Percival. Naive differences of executable code. 2003.

[15] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*. USENIX, 2006.

[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[17] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[18] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[19] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.