Live Debugging of Distributed Systems

Darren Dao[1], Jeannie Albrecht[2], Charles Killian[3], and Amin Vahdat[1]

[1] University of California, San Diego, La Jolla, CA
[2] Williams College, Williamstown, MA
[3] Purdue University, West Lafayette, IN

Abstract. Debugging distributed systems is challenging. Although incremental debugging during development finds some bugs, developers are rarely able to fully test their systems under realistic operating conditions prior to deployment. While deploying a system exposes it to realistic conditions, debugging requires the developer to: (i) detect a bug, (ii) gather the system state necessary for diagnosis, and (iii) sift through the gathered state to determine a root cause. In this paper, we present MaceODB, a tool to assist programmers with debugging deployed distributed systems. Programmers define a set of runtime properties for their system, which MaceODB checks for violations during execution. Once MaceODB detects a violation, it provides the programmer with the information to determine its root cause. We have been able to diagnose several non-trivial bugs in existing mature distributed systems using MaceODB; we discuss two of these bugs in this paper. Benchmarks indicate that the approach has low overhead and is suitable for *in situ* debugging of deployed systems.

1 Introduction

Debugging a distributed system is challenging because its operation depends not only on its internal functions and state, but also on the functions and state of the set of nodes it runs on and the network linking them. At any point in time, correctness depends on a combination of past and present system, node, and network states. Replicating the vast array of possible states and exposing a distributed system to them prior to deployment is not feasible. As a result, many bugs only manifest during deployment, when the "perfect storm" of state transitions trigger them.

Despite recent advancements, most developers still debug distributed systems in an ad hoc fashion by inserting custom print statements to generate output logs, which they parse for errors after an execution ends. Ad hoc approaches require developers to know what to print and what to expect *a priori*, which limits their usefulness for finding unexpected bugs. Existing advanced debugging techniques, while useful, have drawbacks when used for debugging deployed systems. Model checkers force the programmer to define a specification of the system, and then systematically explore a system's state space for violations of the specification. However, the exploration does not capture the vast and complex set of node and network states that impact a deployed system. Replay-based tools, which enable offline analysis of systems, do not detect bugs at runtime, while log-based analysis tools, which systematically process output logs for errors, impose the high overhead of generating and storing log data.

An ideal tool for debugging deployed distributed systems has the following characteristics.

O. de Moor and M. Schwartzbach (Eds.): CC 2009, LNCS 5501, pp. 94–108, 2009.

- **Easy to Use.** The tool should hide low-level implementation details from the developer so it is easy to understand, as well as automate common tasks to minimize its impact on the standard development process.
- **Powerful.** The tool should be powerful and flexible enough to assist programmers in finding a wide variety of bugs in different distributed systems.
- **Low Overhead.** Since many bugs do not manifest until deployment, the tool must operate on deployed systems. Low overhead is essential for using on a deployed system without degrading its performance.

We designed and built MaceODB, an online debugging tool for the Mace [1] language, to satisfy these characteristics. Using MaceODB, we were able to find non-trivial bugs in existing mature distributed systems using only a small amount of additional information provided by the developer. Our performance evaluation shows that MaceODB has little impact on the performance of the systems under test.

The rest of this paper is organized as follows. Sections 2 and 3 detail the design and implementation of MaceODB. Section 4 reports on our experiences using MaceODB and Section 5 reports on its performance. Finally, we review related work in Section 6 and conclude in Section 7.

2 Design of MaceODB

Mace [1] 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. Imposing structure and restrictions on application development allows Mace to support high-level debugging techniques, including efficient model checking and causal-path analysis [2]. The limitation of the Mace model checker is its inability to debug live systems: MaceODB addresses this limitation.

MaceODB is an extension to the Mace compiler that adds new instructions for translating developer-defined system properties into code that checks for property violations at runtime. To use MaceODB, the programmer adds liveness and safety properties to their Mace application (see Figure 1). After specifying a set of properties, the Mace compiler generates the application's C++ implementation from its specification. The compiler invokes MaceODB to parse the developer-defined properties and adds additional code that checks for property violations at runtime. During execution, the property-checking logic automatically reports violations back to the programmer.

2.1 Properties

MaceODB extends Mace by allowing programmers to define properties for their distributed systems. These properties are predicates that must hold true for some subset of the participating nodes in the system. (Note that predicates are always "const" functions, and thus cannot have side-effects on the state of the nodes.) MaceODB currently supports two types of properties: safety properties and liveness properties.

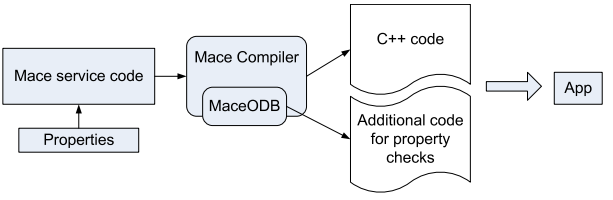


Fig. 1. Overview of MaceODB

Safety properties. Safety properties are predicates that should always be true. These properties assert that the program will never enter an unacceptable state [3]. Formally, they can be expressed as statements of the form *always p*, where *p* is predicate that must be evaluated to true at all times. For example, suppose we build a peer-to-peer file transfer system that constructs an overlay tree. An important safety property of this system is that there should never be any loops in the underlying topology. When defining safety properties, programmers must know exactly what violations to look for and define unacceptable states in advance.

Liveness properties. Liveness properties are predicates that should eventually be true [3]. For example, in a peer-to-peer file transfer system, all participants should eventually enter the joined state to be part of the overlay tree. Note that liveness properties, unlike safety properties, apply to an entire program's execution rather than individual states. As a result, liveness properties are more difficult than safety properties to evaluate for violations. The benefit of using liveness properties is that they more naturally align with the way developers reason about the state of a system. Defining high-level liveness properties is easier for a developer since they correspond more directly to design specifications defined by the developer.

2.2 Specifying Properties

The most effective way to specify safety and liveness properties is to analyze the correct system behavior under steady-state operation. After identifying the desirable behavior, the programmer writes liveness properties to verify that the desired behavior is upheld throughout an execution. If any liveness violations occur, the programmer can leverage insight from the violations to specify additional safety properties. Although safety properties are more difficult to define *a priori*, these properties contain more specific checks to help narrow down the bugs causing liveness violations. For example, consider a bug in which a certain timer becomes unscheduled and causes a liveness property to fail. After detecting this liveness violation, the programmer adds an additional safety property to ensure that the timer is always scheduled.

To write the safety and liveness properties, programmers use the Mace compiler's grammar. A simplified version of the grammar is in Figure 2. We have used this grammar to write safety and liveness properties for several existing Mace applications and services (see Table 1). We found that most properties permit concise specifications consisting of a few lines of code. For example, consider the property "AllJoined" in Table 1. "AllJoined" is an example of a liveness property of the Mace RandTree service, which is a simple distributed application that constructs a random overlay tree. The purpose of

```
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}
```

Fig. 2. Simplified grammar for writing MaceODB safety and liveness properties

"AllJoined" is to check that all participating nodes eventually enter the `joined` state that signals a valid connection to the overlay. In this case, we express the property in a single line of code.

Now consider the "Timer" property in Table 1. In "Timer," `recovery` is a timer object defined by the RandTree developer (see Section 4.1). The timer object has a `nextScheduled()` method that returns the next time the recovery process will execute; the purpose of the property is to verify that once each system participant completes the `init` state it executes the `recovery` timer. The `recovery` timer ensures that subsequent failures trigger the recovery process.

2.3 Centralized Property Evaluation in MaceODB

Our initial design of MaceODB uses a centralized approach for evaluating the properties at runtime. The design uses a central server that is responsible for evaluating all the properties across the entire system (see Figure 3). The design consists of two components: the Data Exporter module and the Property Checking module. The Data Exporter module operates on each node in the system, extracts data that describes the execution's current state, and forwards the timestamped data to the central server. The central server then uses the Property Checking module to evaluate the data's liveness and safety properties. Upon receiving data from each Data Exporter module, the central server invokes the Property Checking module to perform the property evaluation, and generates a report of property violations.

To understand the Data Exporter and Property Checking modules, consider the Pastry [4] (a typical Distributed Hash Table) property "LeftRight" in Table 1. This property compares the size of `myleafset` to the size of `myleft` plus the size of `myright`. The data of interest is `myleafset.size()`, `myleft.size()`, and

Table 1. Examples of properties that are used in Mace applications (Pastry, Chord, RandTree)

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`
	`};`

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)`
	`};`

AllJoined (RandTree)	*Test that eventually all the nodes will join the system.* `\forall n \in \nodes : n.state = joined;`

`myright.size()`; the Data Exporter module extracts these attributes from each participating node and sends them to the central server. The Property Checking process consists of iterating through the data and evaluating whether `myleafset.size()` is equal to `myleft.size()` + `myright.size()`; if they are not equal, the evaluation process returns false, indicating a property violation.

Our performance evaluation shows that the centralized approach is sufficient for most properties. However, some properties require each participating node to send large amounts of data to the central server, creating a bottleneck that slows down the property evaluation process and reduces overall system performance. We include optimizations to reduce the data sent using a binary diff tool [5] to compare the differences between snapshots of forwarded data. The optimization significantly reduces the bandwidth, although the central server remains a bottleneck for sufficiently large systems (*i.e.*, more than 100 nodes) due to the memory and processor time required to process diffs.

2.4 Decentralized Property Evaluation in MaceODB

In this section we describe a decentralized design for MaceODB to address problems with the centralized design. The decentralized design also uses the Data Exporter and

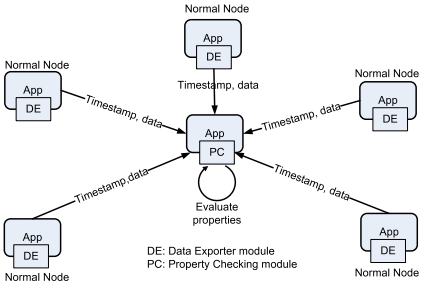


Fig. 3. MaceODB centralized design

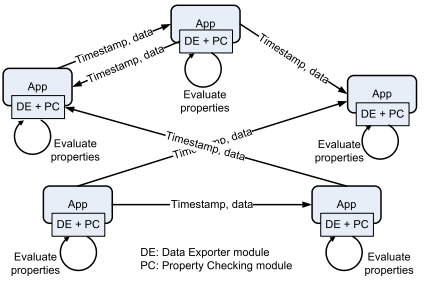


Fig. 4. MaceODB decentralized design

Property Checking modules. However, unlike the centralized approach, the Property Checking module is now present in all system nodes (see Figure 4), requiring each node to be responsible for evaluating their individual properties. The design eliminates the central server bottleneck, and eliminates the single point of failure. Additionally, we use a membership service to address network and node failures.

In decentralized MaceODB, we represent properties as dataflow graphs. Figure 5 shows an example of this representation for the "LeftRight" property described in Table 1. Dataflow graphs consist of three main components: the leaves, the vertices, and the arcs. The leaves correspond to the data used to evaluate properties, which come from the local node performing the evaluation or from the Data Exporter modules on other nodes. The vertices represent the operations that evaluate the properties. Together, these operations form the basis for the Property Checking module. The arcs represent the input/output flows, and describe the dependencies between the operations.

At runtime, each node generates instances of these graphs for each timestamp, and evaluates vertices of the graphs as soon as upstream inputs are available. The Property Checking module processes each vertex and evaluates vertices that are ready, and evaluates vertices of different timestamps simultaneously in a pipelined fashion. Exploiting the property-level parallelism demonstrates a key benefit of representing properties as dataflow graphs. The representation separates the data, operations, and input/output dependencies into independent blocks that are evaluated simultaneously in parallel.

Note that, in both the centralized and decentralized approach, the Data Exporter and the Property Checking modules are automatically generated by MaceODB. Developers do not write any additional code—only properties—and MaceODB generates all the low-level code for exporting state data and evaluating properties.

2.5 Globally Consistent Snapshots

For both centralized and decentralized designs, many MaceODB properties must be evaluated across all participating nodes. In order to evaluate these properties, we need a consistent snapshot of the state of the entire system. To support this, we added a logical clock [6] to the Mace language. Each node in the system maintains its own logical clock that 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

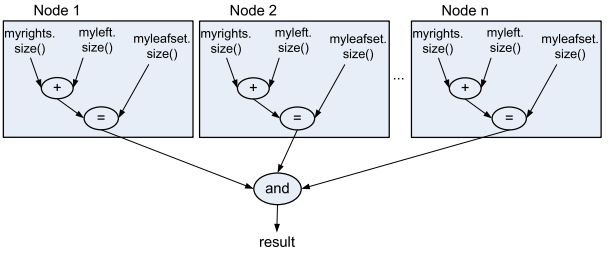


Fig. 5. Example of using a dataflow graph to represent the Pastry LeftRight property

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, establishing the happens-before relationship. Using this mechanism, MaceODB associates each node's data with a global timestamp, thus providing a globally consistent snapshot.

3 Implementation of MaceODB

This section discusses the centralized and decentralized implementations of MaceODB.

3.1 Centralized Implementation

The two key components of the centralized approach are the central Property Checking module and the distributed Data Exporter module. We now describe how MaceODB constructs these modules in detail.

Data Exporter Module. As described in Section 2.3, the Data Exporter module is responsible for sending data from participating nodes back to the central server for property evaluation. Thus, an important task in building the Data Exporter module is determining what data to send. In the simplest implementation, MaceODB uses the grammar specification in Figure 2 to parse through each property, and identify all the variables associated with that property. Each variable's value corresponds to the data sent to the central server. While this works well for most properties, there are edge cases that are inefficient or that the approach fails to cover.

First, consider how the simple implementation is inefficient. For the "Timer" property in Table 1, MaceODB identifies the following variables as exported data: `n.state`, `init`, `n.recovery.nextSchedule()`, and `0`. Exporting this set of variables is inefficient since they are either constants or variables that come from the same node. As a result, it is possible to evaluate "Timer" without exporting any data. Therefore, instead of sending data to the central server, each node evaluates the property *locally*, and forwards only the result. The optimization reduces the data transmitted to the server. "LeftRight," "Timer," "PredNotNull," and "AllJoined" are all examples of local properties.

The simple implementation fails to work for properties that include methods with parameters that require data from other nodes. In this case, MaceODB requires each node to export the results from executing the method calls to the central server. Unfortunately, the nodes are not capable of completing the method calls independently, since

they require data from peers. To solve this problem, each node sends its local state to the central server. The server then deserializes the state and constructs dummy objects for each node. Using these dummy objects, the server simulates method calls using the appropriate parameter inputs.

After determining the data to send, the next phase in building the Data Exporter module is to generate the actual code for sending the data. We leverage functionality provided directly by Mace's `Message` class, which allows programmers to write code for sending messages between peers.

Property Checking Module. The central server uses the Property Checking module to evaluate properties and alert the programmer of property violations. The module's primary task is to evaluate each property by parsing and breaking them into smaller expressions as specified by the grammar in Figure 2. These expressions correspond directly to the operations required in order to evaluate the properties. For example, consider again the "Timer" property in Table 1. When MaceODB parses that property, it identifies the following expressions/operations:

- Equation Expression #1 - Compares `n.state` and `init`.
- Equation Expression #2 - Compares `n.recovery.nextScheduled()` and `0`.
- Join Expression - Performs a logical OR operation on the results of #1 and #2.
- Quantification Expression - Performs a `forall` loop operation.

After identifying the above expressions, MaceODB generates a C++ method for each expression. Together, these methods form the complete Property Checking and Data Exporter modules that run on the central server.

3.2 Decentralized Implementation

The decentralized implementation also includes Property Checking and Data Exporter modules, but the implementations are different due to the distributed design. This section discusses the key differences in the construction of these modules.

Data Exporter Module. In the decentralized implementation, each node uses its own Data Exporter module to exchange data with other nodes. MaceODB generates two classes to accomplish this in the decentralized approach: `RequestMessage` and `ReplyMessage`. These classes extend the Mace `Message` class. As their names imply, the `RequestMessage` class is used for requesting data, and the `ReplyMessage` class is used for returning the result.

To understand the approach, consider the "KeyMatch" property in Table 1. The property has an equality operation that compares `range.first` and `range.second`. The `range.second` input comes from the node executing the operation. The `range.first` input comes from the node specified by the result of `n.getNextHop(n.range.second, -1)`. Until that input is available, it is impossible to execute the equality operation. As a result, 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 a `ReplyMessage`, which contains the requested data `range.first`. Upon receiving `ReplyMessage`, the original node extracts the returned data, and uses it as input in the equality operation.

Property Checking Module. The Property Checking module in decentralized MaceODB consists of operations that provide instructions for evaluating properties. In terms of the dataflow graph representation, these operations correspond to the vertices of the graphs. To generate code for the vertices, MaceODB parses the properties and identifies all operations required to check the properties. For each operation found, MaceODB then generates a C++ class to represent it. Note the difference from the centralized approach, where we generate methods instead of classes. Using classes is a more flexible and object-oriented technique for constructing 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.

At runtime, instances of these classes are created and stored in a queue. 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 Experiences Using MaceODB

We have used MaceODB to test a variety of systems implemented in Mace, including RandTree, Pastry [4], Chord [7], Scribe [8], SplitStream [9] and Paxos [10]. Most of these systems are mature, stable, and have been tested extensively in the past. However, using MaceODB we were still able to find non-trivial bugs in RandTree and Chord.

4.1 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 [11] and RanSub [12]. An important liveness property that RandTree must hold is that there should be only one overlay tree that includes all participating nodes. In case of network and node failures, this property does not always hold. To address this issue, a recovery timer was added that periodically checks to see if there was a network partition, and invokes the recovery process as needed. Using "Timer" in Table 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, the state was not `init` and the recovery timer was not set. Using that knowledge, we went back to the source code for RandTree and checked where the recovery timer was scheduled. Figure 6 shows an excerpt of the code that contains the bug.

The most obvious problem with the code shown in Figure 6 is that `recovery.reschedule(TIMEOUT)` never gets called if `peers` is empty. To fix the bug, we moved that statement out of the else block. Thus 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

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

Fig. 6. RandTree bug found using MaceODB

node joins the system, it always joins together with another peer. Hence, when the code above is executed, `peers.empty()` will always return false, causing the execution flow to go directly to the else block and the recovery timer to be scheduled. This is an example of the limitations of using model checking: the checking is done in a specialized environment, which can be quite different from the environment in which the real system is deployed. This 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.

4.2 Chord

Chord is a P2P distributed lookup protocol that supports a single operation of mapping keys to nodes [7]. Using Chord, the participating nodes create a ring topology. Each node in this ring has pointers to its successor and predecessor nodes in the ring. To join the ring, a new node obtains 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 that runs to ensure that global successor and predecessor pointers are consistent and correct.

The predecessor pointers are used in the lookup and stabilize process in Chord, and it is important that they are updated correctly even in the presence of node churn and failures. To test our implementation of Chord, we use the "PredNotNull" liveness property (see Table 1) to check that eventually, all the predecessor pointers are not null. This minimal check ensures 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 one remaining node will become null. It will eventually fix itself if the failed nodes recover. However, if they remain down indefinitely, 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.

Table 2. Impact on goodput when 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%

5 Performance Evaluation

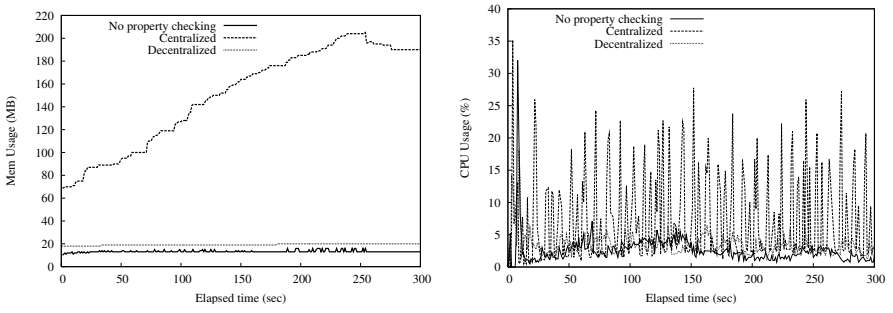
To evaluate MaceODB's performance, we performed a macro-benchmark that measures its overhead. Our results show that MaceODB is lightweight, and incurs minimal overhead on systems. Additionally, we performed a micro-benchmark to quantify the time to evaluate different types of properties.

5.1 MaceODB Overhead

In this section we analyze the impact that MaceODB incurs on Macedon, a data streaming application that can be run on top of any multicast or unicast service. For our experiments, we run Macedon on top of RandTree, Scribe, and SplitStream. For each of these services, we run Macedon with and without MaceODB, and compare the differences in goodput, memory usage, and CPU usage.

We ran our experiments on several different network topologies that range from small systems of 25 clients to larger systems of 100 clients. These clients are emulated on 17 physical machines with the ModelNet network emulator [13]. Each physical machine has a dual-core Xeon 2.8 MHz processor with 2GB of RAM. The emulated topologies consist of an INET network with 5000 total nodes. The emulated clients have bandwidths ranging from 6,000–10,000 Kbps, and latencies ranging from 2–40 ms.

Goodput and Scalability. To evaluate goodput, we measured the number of packets Macedon sent and received for a specific timeframe (5 minutes). We then calculated the impact that MaceODB incurs on the system's goodput (useful throughput) by plotting the ratio of the loss in goodput when using MaceODB to the goodput of the system when run without MaceODB. We present the results in Table 2. Based on these results, we see that MaceODB performs well for systems with 50 nodes or less. For these systems, the impact in most cases is less than 7%. The results are quite different in larger systems. Using the centralized design, the performance is poor when running on Scribe and SplitStream. The impact on the goodput is as high as 13.8% for systems of 75 nodes



(a) Memory usage of Macedon. (b) CPU usage of Macedon.

Fig. 7. Memory and CPU usage of Macedon with and without MaceODB

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. The performance of the decentralized design is much better. Its impact on the goodput is approximately 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.

Memory Usage. Besides measuring the goodput, we also measured memory and CPU usage. Figure 7(a) shows the results of memory usage during the 5 minute Macedon experiments. Without MaceODB, Macedon used approximately 18 MB of RAM. Using this as the base value, we compared it against the memory usage of Macedon when it was run with MaceODB enabled. Our results show that depending on how we evaluated the properties, the impact on memory usage was quite different. With the decentralized design, the memory usage was around 20 MB, which is slightly higher than the base value, but still acceptable. In the centralized design, the memory usage on the central server is significantly higher. Even with a special memory cleanup mechanism enabled, the central server still used 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 is forwarded from the other nodes. This data is stored in memory and cannot be deleted until it forms a complete snapshot of the whole system. At that point, the central server evaluates the properties using the newly created snapshot and removes the old data.

CPU Usage. Figure 7(b) shows the CPU usage during the 5 minute Macedon experiments. Without MaceODB, the CPU usage fluctuates around 2–3%. With decentralized MaceODB, the CPU usage is only slightly higher. 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 were as high as 28%. This further confirms the fact that the centralized approach is not scalable.

Summary. 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 at least 100 nodes. 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 having a significant impact on the system's performance.

5.2 Evaluating Different Types of Properties

Next we measured the time required to evaluate different types of properties in decentralized MaceODB. The goal was to identify properties that are expensive to evaluate. Similar to the macro-benchmark, we ran our experiments on ModelNet, and this time we ran 100 instances of Macedon on top of RandTree and Pastry. We modified Macedon to log the time before and after each property evaluation. The experiments were then run for 5 minutes. At the end of each run, we processed the logs, and calculated the average time for each property evaluation. The average times were as follows: LeftRight/Pastry 79 μs, KeyMatch/Pastry 210 ms, Timer/RandTree 60 μs.

These results show that the cost of evaluation varies greatly among the properties. To understand the reason behind this variance, consider how the properties are evaluated. The properties with the shortest times 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" and "Timer." With this type of property, the cost of evaluation is a function of how fast the CPU can process each operation. For our particular setup, this value is approximately 70 microseconds on average.

Now consider properties that are more expensive to evaluate, such as "KeyMatch." Expensive properties contain operations that require 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, so that the speed of evaluating the 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 aligns with our results since the cost of evaluating "KeyMatch" is slightly more than 200 ms.

6 Related Work

There are several related techniques for debugging distributed systems. We group these techniques into four categories: model checking, replay-based checking, log analysis, and on-line debuggers.

Model Checking. Prior work has proposed model checking as a mechanism for debugging distributed systems [2]. With model checking, the programmer defines system specifications, and uses the model checker to systematically explore the state-space of the system while checking for specification violations. The approach is a powerful debugging tool since it is possible to traverse large state-spaces in small timeframes,

allowing the programmer to discover difficult-to-find bugs. However, the checking process is typically done in a controlled and virtualized environment that does not accurately reflect the deployment environment. MaceODB addresses the limitation and is able to detect bugs that appear during deployment.

Replay-based Checking. Much research has gone into replay-based checking where the programmer has the ability to replay a program while replicating its order and environment. A notable example of replay-based checking is *liblog* [14], which addresses large distributed systems. Liblog logs the execution of deployed application processes, and allows programmers to replay them deterministically. The benefit of using *liblog* or any replay tool is the ability to consistently reproduce bugs from previous executions. While the capability enables offline analysis, its weakness lies in the high cost of logging and replaying an entire execution, especially for large systems. However, MaceODB and replay tools are complementary. A programmer may use MaceODB to detect runtime bugs, and then use replay-based checking for offline analysis.

Log Analysis. Many systems focus on parsing through logs to perform postmortem 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. Pip then provides the programmer with the capability to query the logs and a visual interface for exploring the expected and unexpected behavior. The main problem with Pip, and many other log-based analysis tools, is the high overhead incurred by logging the data.

Other On-line Debuggers. MaceODB is similar in spirit to D^3S [16]. Both share the ideas of using predicates and representing predicates using dataflow graphs. MaceODB differs from D^3S in the way predicates are written. D^3S predicates are written in a mixture of C++ and a scripting language, requiring programmers to specify the stages and the input/output dependencies. MaceODB predicates are written in the Mace language, and the compiler generates the C++ code to represent the stages and dependencies. As a result, MaceODB is easier to use. However, in some cases MaceODB is less efficient than D^3S since MaceODB has the potential to send more data than what is necessary. CrystalBall [17] is a concurrently developed extension of MaceMC [2] with similar goals as MaceODB. Both perform property checking of live running distributed systems. CrystalBall focuses on looking forward during execution, where MaceODB focuses on minimizing overhead and the distribution of property checking.

7 Conclusions

Debugging distributed systems is a challenging task. In this paper we present MaceODB, a tool that makes the task easier by providing programmers with the ability to perform online property checking for services written in Mace. MaceODB is easy to use, yet flexible and powerful enough to catch several non-trivial bugs in the existing Mace services. Our results show that MaceODB tolerates node and network failures inherent to distributed systems, and has low overhead, which makes it possible to use on live systems without significant performance degradation.

References

1. Killian, C.E., Anderson, J.W., Braud, R., Jhala, R., Vahdat, A.M.: Mace: Language Support for Building Distributed Systems. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (2007)
2. Killian, C.E., Anderson, J.W., Jhala, R., Vahdat, A.: 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)
3. Kindler, E.: Safety and Liveness Properties: A Survey. Bulletin of the European Association for Theoretical Computer Science 53 (1994)
4. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, p. 329. Springer, Heidelberg (2001)
5. Percival, C.: Naive Differences of Executable Code (2003)
6. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7) (1978)
7. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. IEEE/ACM Transactions on Networking 11(1) (2003)
8. Rowstron, A.I.T., Kermarrec, A.-M., Castro, M., Druschel, P.: SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In: Networked Group Communication (2001)
9. Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., Singh, A.: SplitStream: High-Bandwidth Multicast in Cooperative Environments. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2003)
10. Lamport: The Part-Time Parliament. ACM Transactions on Computer Systems 16 (1998)
11. Kostić, D., Rodriguez, A., Albrecht, J., Vahdat, A.: Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (2003)
12. Kostić, D., Rodriguez, A., Albrecht, J., Bhirud, A., Vahdat, A.: Using Random Subsets to Build Scalable Network Services. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS) (2003)
13. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., Becker, D.: Scalability and Accuracy in a Large-scale Network Emulator. In: Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI) (2002)
14. Geels, D., Altekar, G., Maniatis, P., Roscoe, T., Stoica, I.: Friday: Global Comprehension for Distributed Replay. In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2007)
15. Reynolds, P., Killian, C.E., Wiener, J.L., Mogul, J.C., Shah, M.A., Vahdat, A.: Pip: Detecting the Unexpected in Distributed Systems. In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2006)
16. Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M.F., Zhang, Z.: D^3S: Debugging Deployed Distributed Systems. In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2008)
17. Yabandeh, M., Knežević, N., Kostić, D., Kuncak, V.: CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. Technical report, School of Computer and Communication Sciences, EPFL, Switzerland (2008)