Mace: Language Support for Building Distributed Systems

Charles Killian James W. Anderson Ryan Braud Ranjit Jhala Amin Vahdat

University of California, San Diego

{ckillian,jwanderson,rbraud,jhala,vahdat}@cs.ucsd.edu

Abstract

Building distributed systems is particularly difficult because of the asynchronous, heterogeneous, and failure-prone environment where these systems must run. Tools for building distributed systems must strike a compromise between reducing programmer effort and increasing system efficiency. We present *Mace* , 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. Because Mace programs compile to C++, programmers can use existing C++ tools, including optimizers, profilers, and debuggers to analyze their systems.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Specialized application languages

General Terms Languages, Reliability, Performance

Keywords Mace, domain specific languages, model checking, debugging, distributed systems, concurrency, event driven programming

1. Introduction

Designing and implementing robust and high-performance distributed applications remains a challenging, tedious, and error-prone task. Currently, there are three ways of specifying distributed systems. First, formalisms such as I/O Automata [24] or the Pi-Calculus [26] can be used to model distributed algorithms as collections of finite-state automata (or processes), one for each node of the system that interact by sending and receiving messages. Though these formalisms succinctly capture the essence of many distributed protocols and algorithms, they abstract away and ignore the low-level implementation details essential to deploying robust, high-performance systems. Second, higher-level programming languages such as Java, Python, and Ruby have eased some of the tedium associated with building distributed systems. However, they often introduce performance overheads and do not significantly simplify the task of ensuring system correctness or identifying inevitable performance problems.

Thus, developers seeking efficiency resort to the third option of assembling applications in an ad-hoc, bottom-up manner. While the resulting systems may be fast and reliable, they sacrifice structure, readability, and extensibility. The lack of structure in particular significantly limits the ability to apply automated tools, such as model checkers, to find subtle performance and correctness problems.

Our goal is to determine whether programming language, compiler, and runtime support can combine the elegance of high-level specifications with the performance and fault-tolerance of low-level implementations. We seek to drastically lower the barrier to developing, maintaining, and extending robust, high-performance distributed applications that are readable and amenable to automatic analysis for performance and correctness problems.

The main difficulty in building these systems arises from the distributed, concurrent, asynchronous, and failure-prone environment where distributed systems run. System complexity requires that applications be *layered* on top of fast routing protocols, which are built on top of efficient messaging layers. *Concurrency* and asynchrony imply that events simultaneously take place at multiple nodes in unpredictable orders. Messages may be delivered in arbitrary orders, dropped or delayed nearly indefinitely. Nodes may at any time have multiple outstanding messages in-flight to other nodes and multiple pending received messages ready to be processed. Further, an arbitrary subset of nodes or links may *fail* at any time, leaving the system as a whole in a temporarily inconsistent state.

These properties make maintaining performance and correctness difficult. A single high-level system request may require communication with many nodes spread across the Internet. Often, even seemingly correct distributed system implementations perform an order of magnitude more slowly than expected. *Analyzing* executions to find the source of such problems frequently reduces to searching for a needle in a haystack: among (at least) millions of individual message transmissions, algorithmic decisions, and the large number of participating nodes, which network link, computer, or low-level algorithm resulted in performance degradation?

While each of these problems has well-known solutions, the task of addressing them simultaneously proves to be quite challenging because of their subtle interactions. For example, object-oriented design is the canonical way to build systems from subsystems, but for distributed systems hiding internal state from other layers results in serious performance penalties and duplicate effort. Similarly, there are standard ways to detect and handle failures, but the code for doing so must be interspersed (usually repeatedly at multiple points) with the code for handling common case operation, not only obfuscating the code but also eliminating the high-level structure required to use techniques like model checking [13] and performance debugging [2, 4, 8].

In this paper, we present Mace, a new C++ language extension and source-to-source compiler for building distributed systems in C++. Mace seamlessly combines *objects*, *events*, and *aspects* to simultaneously address the problems of layering, concurrency, failures, and analysis. While these are well known Programming Language ideas, the key advances of Mace are twofold. First, we unify

PLDI'07 June 11–13, 2007, San Diego, California, USA.

in one development environment the diverse elements required to build robust and high-performance distributed systems. Second, and more importantly, by defining a *language extension* to write distributed systems, we are able to *restrict* the ways that such systems can be built. For our domain, this restriction is both *expressive* enough to permit the compilation of readable high-level descriptions into implementations matching the performance of hand-coded implementations and *structured* enough to enable the use of automatic, efficient, and comprehensive static and dynamic analysis to locate and understand behavioral anomalies in deployed systems.

By using a structured—yet expressive—approach tailored to distributed systems, Mace provides many concrete benefits:

- Mace allows the programmer to focus on describing each layer of the distributed system as a reactive state transition system, using events and transitions as the basis for system specification. This explicitly maintains structure given by high-level formalisms while enabling high-performance implementations.
- Mace uses the semantic information embedded in the system specification to automatically generate much of the code needed for failure detection and handling, significantly improving readability and reducing the complexity of maintaining internal application consistency.
- Mace supports automatic profiling of individual causal paths—the sequence of computation and communication among nodes in a distributed system corresponding to some higher level operation, e.g., a lookup in a distributed hash table. Mace exports a simple language that allows developers to match their expectations of both system structure and performance against actual system behavior, thereby isolating performance anomalies.
- Mace's state transition model enables practical model checking of distributed systems implementations to find both safety and liveness bugs. We built a model checker, MaceMC, to successfully find subtle bugs in a variety of complex distributed systems implementations. Most of the bugs were quite insidious, present in mature code, and could not be found without exploiting the structure preserved by Mace.

Mace is fully operational, has been in development for four years, is publicly available for download, and has been used by researchers at UCSD, HP Labs, MSR-Asia, and a handful of universities worldwide in support of their own research and development. We have implemented more than ten significant distributed systems in Mace, most of which were originally proposed by others. This set includes Distributed Hash Tables [30, 32, 35], Application Layer Multicast [7, 15, 20], and network measurement services [9] ready to run over the Internet. Along with a summary of our implementations, we present a detailed performance comparison between the publicly available version of Bamboo [30], a state-of-the-art DHT, and a Mace implementation we wrote from scratch. Mace tools for isolating performance and correctness errors significantly ease the task of debugging complex distributed systems. By combining the outlined benefits, we have built Mace implementations—in an order of magnitude less code—that outperform the original systems.

2. Overview

Drawing from our experience building a variety of distributed systems based on high-level specifications, we categorize the gap between specification and low-level features essential to real deployments into the following categories:

- **Layers:** To manage complexity, network services consist of a hierarchy of layers, where higher level layers are built upon lower levels. The canonical example is the Internet protocol stack where, for example, the physical layer is responsible for modulating bits on a medium, the link layer delivers packets from one node to another on the same physical network, the network layer delivers packets between physical networks, and the transport layer provides higher level guarantees such as reliable, in-order delivery. Each layer builds upon well-defined functionality of the layer below it and can typically work on a variety of implementations of the underlying layer's interface.
- **Concurrency:** A distributed implementation must properly contend with and exploit concurrency to maximize performance. For example, an overlay routing application must simultaneously contend with the application layer sending requests to the routing layer, the networking layer receiving new messages that must be passed up to the routing layer, and timers executing scheduled tasks. While these events may be interleaved on a single node, they can also be arbitrarily interleaved across nodes as well.
- **Failures:** A robust implementation must account for the inevitable failures of different components or nodes of the distributed system. Failures are often difficult to detect; for instance it is impossible to distinguish between a failed node and one that is particularly slow. Further, the remaining nodes must correctly update their state to reflect the new configuration, lest inconsistencies lead to further errors.
- **Analysis:** Given the complex operating environment, there are always performance bottlenecks and correctness issues that arise because the developers overlooked some subtle scenario or miscalculated some parameter like a message timeout. An implementation must be structured and readable enough to permit the manual and automatic analyses required to fix such performance and correctness problems.

While well-known techniques address each of these issues in isolation, the primary challenge in our setting is to devise mechanisms that help the programmer resolve the tensions arising from complex interactions between the four problems. For example, the standard solution to the problem of layering is Object-Oriented design. However, for high-performance applications, treating layers as black boxes that hide their internal state and mask failures leads to performance bottlenecks. For instance, higher level routing algorithms greatly benefit from lower level information about link latencies and knowledge about which nodes or links have failed. Further, multiple sources of concurrency complicate the task of propagating information consistently between layers.

Similarly, failures make it difficult to design a layering mechanism. The approach of masking low-level failures—while appealing because it simplifies higher layers—is insufficient in distributed environments because it sacrifices significant performance gains available from notifying the upper layers of the failure. For instance, the transport layer could mask failures by buffering sent messages and attempting to resend them until it succeeds; however, doing so would prevent higher layers from adjusting their own state to achieve better performance, such as a multicast layer reconfiguring its tree structure for higher throughput after a failure. Unfortunately, the task of notifying the upper layers is complicated by the fact that failures can happen concurrently with other system events. Further, concurrency makes it tricky to cleanly separate the failure detection and handling code from the rest of the common-case code, obfuscating the resulting system and destroying structure.

Finally, standard techniques such as profiling to find performance bottlenecks and model checking to find pernicious bugs typically cannot be applied to distributed systems implementations. In ad-hoc implementations, the code that handles concurrency and failures obscures code structure making manual and automated rea-

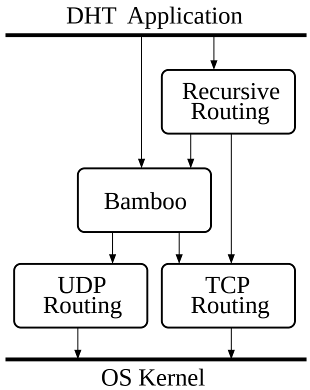

Figure 1. Bamboo DHT design architecture.

soning difficult (but essential). The principle technique used by developers to analyze deployed systems is tedious ad-hoc logging that clutters the code and often delivers only limited value because the programmer must manually stitch together spatially and temporally scattered logs.

Finally, concurrency makes it difficult to reason about or to simply even replicate behaviors (due to non-deterministic factors like network latencies and scheduling decisions), thereby severely increasing the time and effort required to find complex bugs via testing. In our experience, particularly subtle bugs may remain latent for weeks in a deployed system. Further, because these bugs often result from inconsistencies between the state at multiple nodes, the subsequent departure or failure of a node after the bug manifests itself can push the system back into a consistent state, masking the bug and making it even more difficult to track.

Thus, to develop high-performance systems from high-level specifications, we must devise techniques to architect the system, to determine when failures have occurred, and to propagate and exploit information throughout the architecture. These techniques must operate in a concurrent setting, enable modularity and reuseability, and explicate the high-level structure of the algorithm, thereby enabling manual and automatic system level analyses.

Mace Design Principles

To address the challenges posed by the domain of distributed applications, we base Mace on three fundamental concepts.

- **Objects:** Mace structures systems as a hierarchy of *service objects* connected via explicit interfaces. We use an object to implement each layer of the system running on an individual node. The interface for each layer specifies both the functionality *provided* by that layer as well as any *requirements* that must be satisfied to use that layer.
- **Events:** Mace uses *events* as a unified concurrency model for all levels of the system: within an individual layer, across the layers at a single node, and across the nodes comprising the entire system. Each event corresponds to a method implemented by a service object.
- **Aspects:** Mace provides *aspects* to describe computations that cut across the object and event boundaries: in particular, aspects define tasks that need to be performed when particular conditions become satisfied.

While each of these ideas have been studied extensively in isolation, we demonstrate that they combine synergistically to preserve the high-level structure of the distributed system and simultaneously address the complexity and challenges of building robust, high performance implementations. We use a popular Distributed Hash Table (DHT) to illustrate the challenges associated with building distributed systems and our approach to addressing these challenges. DHTs support `put` and `get` operations on a logical hash table whose actual storage is spread across multiple physical machines, and thus form a convenient abstraction for building higher-level applications like distributed file systems [10, 27]. The key properties of a DHT implementation are scalability and robustness to failure. We consider a DHT built on the BAMBOO routing protocol [30] (similar to CHORD [35] or PASTRY [32]).

Layers

Nodes in BAMBOO self-organize into a structure that enables rapid routing of messages using node identifiers. This protocol forms a single layer of the DHT shown in Figure 1. BAMBOO is built on top of a TCP subsystem that maintains network connections and delivers messages and a UDP subsystem that sends latency probes. A recursive routing subsystem routes messages to the node owning a given key by asking BAMBOO for the next hop towards the destination. The DHT application layer uses the lower layers to store and retrieve data.

Mace enables programmers to build layered systems by using objects to implement individual layers and events to facilitate interaction across layers. For each layer, the programmer writes interfaces specifying the events that may be received from or sent to the layers both above and below. A layer's implementation consists of a *service object* that must be able to receive and may send all the events specified in the interfaces. *Thus, Mace combines objects and events to enable programmers to build complex systems out of layered subsystems, thereby abstracting functionality into layers with specified interfaces and allowing the safe reuse of different implementations (meeting the same interface) of a particular layer in different systems.*

Concurrency

In BAMBOO, a key challenge is to provide fast message routing while simultaneously dealing with node churn – *i.e.* the arrival and departure of nodes from the system. To achieve this goal, the system must concurrently process network errors, messages from newly created nodes, and periodically perform maintenance to ensure routing consistency.

In Mace, each service object consists of a *state-transition* system beginning in some initial state. Each node progresses by sequentially processing *external* events originating from the application, the physical network layer, or self-scheduled timers. Upon receiving an event, the service object executes a corresponding *transition* to update its state, during which it may transitively send new events to the layers above and below, each of which are processed synchronously without blocking until completion. Furthermore, a transition may queue new external events locally by scheduling timers and remotely by sending network messages. Once processing for a given external event completes, the node picks the next queued external event and repeats.

The Mace event-driven model provides a unified treatment of the diverse kinds of concurrency that must be handled in an efficient implementation: the reception of messages from other nodes (via the transport layer), the reception of high-level application requests, timers firing, and cross-layer communication all correspond to events that the relevant layers must handle via appropriate transitions. Additionally, Mace ensures that the transitions execute without preemption, freeing the programmer from worrying about exponential interleavings of concurrent executions. Finally, because Mace automatically dispatches events through a carefully tuned scheduler, Mace systems can achieve the throughput necessary for high performance applications with minimal programmer involvement. *Thus, objects, events, and aspects enable Mace to describe each layer of a complex application with the simplicity and conciseness of high-level models; when combined with the modular layering mechanism, Mace provides a succinct representation of the entire computation stack for each node of the distributed system.*

Failures

BAMBOO builds an overlay network forming a logical ring among the nodes. To create and maintain this topology, each node keeps references to its adjacent peers in the ring. If one node fails, the application-level state corresponding to the relationships between that node and its neighbors may become inconsistent, breaking the overlay structure.

Mace uses aspects to cleanly specify how to consistently update local state in response to a variety of cross-layer events, such as node arrivals, departures, and application-level failures. The developer can specify predicates over the variables of a given node that test for programmer-specified inconsistencies. Mace generates code to evaluate the predicate whenever the relevant variables change and to execute the aspect when the predicate is satisfied. Aspects provide an ideal mechanism for specifying and detecting failures and inconsistencies, as without them the developer would have to undertake the tedious and error-prone task of manually placing checking code throughout the system, additionally reducing readability. When a failure occurs, the Mace runtime sends notification events to the appropriate layers. Upon receiving these events, the system executes recovery transitions. *Thus, Mace combines objects, events, and aspects to provide clean mechanisms for* specifying, notifying, *and* handling *various types of failures and for maintaining the consistent internal state necessary for fault-tolerant implementations.*

Analysis

BAMBOO routes messages through several intermediary nodes, so tracing the forwarding path for a specific message manually, for instance to debug the timing of a request, involves inspecting multiple physically scattered log files. Mace simplifies such analysis tasks through preservation of the explicit high-level structure of the distributed application with three techniques. First, Mace uses aspects to separate code needed to log statistics, progress, or debugging information from the actual event handling implementation. By removing the distracting logging statements, Mace keeps the system code readable. Second, Mace exploits the structuring of the computation into causally related event chains to generate event logs, which may be spatially and temporally scattered. These event logs can be *automatically* aggregated into flows describing high-level tasks, extracting the events at individual nodes corresponding to some higher-level operation. The structure preserved in the flows allows developers to use automated analysis techniques to find and fix performance anomalies.

Third, the modular structure of Mace applications enables developers to test the system using simulated network layers that facilitate deterministic replay. We have built a tool, MaceMC, that combines these deterministic layers with a special scheduler that iterates over all possible event orderings. MaceMC systematically explores the space of possible executions to find subtle bugs in the system. The event-driven nature of Mace applications reduces the number of interleavings that must be analyzed, enabling MaceMC to search deep into the execution space. *Thus, objects, events, and aspects combine to structure Mace implementations that enable automated analysis techniques to improve the performance and reliability of the distributed application.*

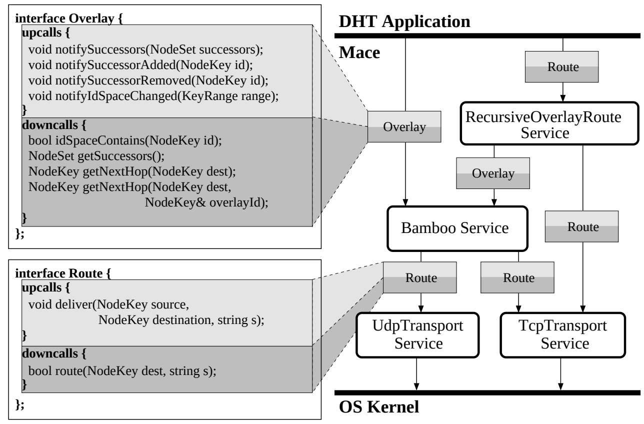

Figure 2. Mace service composition for a DHT application using Recursive Overlay Routing implemented with Bamboo. Shaded boxes (dark for downcall, light for upcall) indicate the interfaces implemented by the service objects.

3. Mace

We now describe the details of how Mace combines objects, events, and aspects to generate high performance, fault-tolerant implementations from high-level specifications. Our C++ language extensions structure each service object as a state machine template with blocks for specifying the interface, lower layers, messages, state variables, inconsistency detection, and transitions containing C++ code implementing event handlers. The template syntax allows the Mace compiler to enforce the architectural design by performing a high-level validation of the service object. Additionally, the structure gives the Mace compiler the necessary information to automatically generate efficient glue code for a variety of tasks that in previous, ad-hoc implementations, had to be manually inserted by the developer. This section discusses how Mace addresses many of the challenges associated with building distributed systems.

3.1 Layers

To specify a distributed system in Mace, the programmer simply specifies the set of layered *service objects* (abbreviated to services) that comprise a single node and the implementation of the required interface for each service object. Figure 2 depicts a more detailed view of the BAMBOO architecture (shown earlier in Figure 1), including the interfaces.

Interfaces. An *interface* comprises a set of *downcall* events and a set of *upcall* events. Upper layers send downcalls received by lower layers. Lower layers send upcalls received by the upper layers. We model events using methods – sending corresponds to calling the appropriate method, and receiving corresponds to executing the method. In Figure 2 on the left, we show two interfaces: `Overlay` and `Route`. For each interface, the top half (lightly shaded box) corresponds to the upcall events, and the lower half (darkly shaded box) shows the downcall events.

Architecture. Developers layer service objects implementing higher layers on top of service objects implementing lower layers. To facilitate modular design and seamless replacement of one service object with another, we specify for each service object the set of lower-level interfaces it *uses* and the upper-level interface it *provides*.

- **Used Interfaces:** When specifying a service, the developer declares each lower-level service with a name and an interface. The service may send any of the downcall events specified in the interface to any of the lower layers, and it must implement all the upcall events to receive any callbacks.

BAMBOO uses two lower-level services of type `Route`, which it binds to local names `TCP` and `UDP`. The BAMBOO implementation can directly call `TCP.route` (resp. `UDP.route`) on the `TCP` (resp. `UDP`) service object implementing the lower level, as `route` is a downcall event in the used interface. Similarly, the lower-level TCP and UDP services can invoke the `deliver` callback on BAMBOO, as it is an upcall in the `Route` interface.

- **Provided Interface:** When writing a service, the developer specifies how upper layers can use the service via a provides interface. The service must implement all downcall events specified in the provides interface and may also send callback events to upper layers, typically in response to some prior request.

  Figure 2 shows that all arrows pointing to BAMBOO have type `Overlay`, indicating that BAMBOO provides the `Overlay` interface to upper-level services. Thus, the BAMBOO service object must be able to receive the `getNextHop` event from the upper layers, as it is a downcall event in the `Overlay` interface. Likewise, the BAMBOO service may send an upcall `notifyIdSpaceChanged` event, which must be implemented by any upper layers using BAMBOO.

Static Checking. The Mace compiler performs two compile time checks to enforce that each service object meets the requirements of the interfaces that it uses and provides. First, the compiler checks that the object implements methods corresponding to all the upcall events in the used interfaces and the downcall events in the provided interface. Second, the compiler checks that the object only calls methods corresponding to downcall events in the used interfaces, and the upcall events in the provided interface.

The service specification explicitly names lower-level services because this knowledge is required to build the service. For example, BAMBOO requires two transports: one for sending protocol related messages (`TCP` by default) and one for probing (`UDP` by default). However, any upper layers that use a given service are unknown when specifying the service, hence those need not and cannot be explicitly named. We observe that the upcall events sent to upper level services are in response to previous requests made by those services. Thus, the Mace compiler automatically generates code such that every downcall is accompanied by a reference to the source of the downcall, and the service employs this reference to determine the destination of the subsequent upcall.

By explicitly decomposing the whole system using layers and interfaces, Mace allows implementations of subsystems to be easily reused across different systems, as any service implementation that meets the statically checked interface specifications can be used as the subsystem. For example, our DHT application works equally well by replacing the BAMBOO service object with service objects implementing the CHORD or PASTRY algorithms, which also provide the `Overlay` interface.

3.2 Concurrency

The standard way of modeling distributed algorithms at a high-level is with state-transition systems. Mace enables developers to reap the many benefits of this structured approach by requiring them to specify each service object as a state transition system where the transitions represent the execution of the methods corresponding to received events. Given specifications for individual service state machines, Mace can automatically compose layers to obtain an efficient, structured system implementation. A state machine specification comprises two basic entities: states and transitions.

```
states { init; preJoining; joining; joined; }
state_variables {
  NodeKey myhash;
  leafset myleafset;
  KeyRange range;
  Table mytable;
  timer global_maintenance __attribute((recur(MAINTENANCE_TIMEOUT)));
  timer join_timer;
}
transitions {
  /* Other transitions . . . */
    scheduler global_maintenance()
      guard (state == joined){
        NodeKey d = myhash;
        for(int i = randint(ROWS); i < ROWS; i++) {
          d.setNthDigit(randint(COLS), B);
        }
        NodeKey n = make_routing_decision(d);
        TCP.route(n, GlobalSample(d));
      }
    upcall forward(const NodeKey& src, const NodeKey& dest,
                   NodeKey& nextHop, const GlobalSample& msg)
      guard (state == joined){
        nextHop = make_routing_decision(msg.key); return true;
      }
    upcall deliver(const NodeKey& src, const NodeKey& dest,
                   const GlobalSample& msg)
      guard (state == joined){
        TCP.route(src, GlobalSampleReply(msg.key, myhash));
      }
    upcall deliver(const NodeKey& src, const NodeKey& dest,
                   const GlobalSampleReply& msg)
      guard (state == joined){
        update_state(src, msg.destHash, true/*known live*/, true/*do probe*/);
      }
}
```

Figure 3. States and Transitions for BAMBOO Service Object

States. States are a combination of the finite high-level control states of the service protocol, along with the (possibly infinite) data states corresponding to values taken by variables such as routing tables, peer sets, and timers. Figure 3 shows how the programmer specifies the high-level states and state variables of the BAMBOO service. The finite high-level states, `init`, `preJoining`, `Joining`, and `Joined` correspond to the four stages of joining the system. The state variables `myhash`, `myleafset`, `mytable`, and `range` correspond to the node's unique identifier, the set of peers and routing table maintained by the node, and the space of keys assigned to the node. In addition, BAMBOO uses two timers, one of which is automatically rescheduled at `MAINTENANCE_TIMEOUT` intervals by Mace compiler generated code.

Transitions. There are three kinds of transitions and corresponding events: upcalls received from lower layers, downcalls received from upper layers, and scheduler events received from self-scheduled timers. Methods implement the transitions and update the state upon receipt of the corresponding event. Figure 3 shows different kinds of transitions corresponding to events the BAMBOO service object may receive. A keyword labels each method and indicates its transition type. Each transition method can be guarded by a predicate over the state variables. This condition may reference the current high-level service state, service state variables, or event parameters. The transition only fires if the guard is true.

To understand how the programmer structures the code for each service into events and transitions, consider the high-level

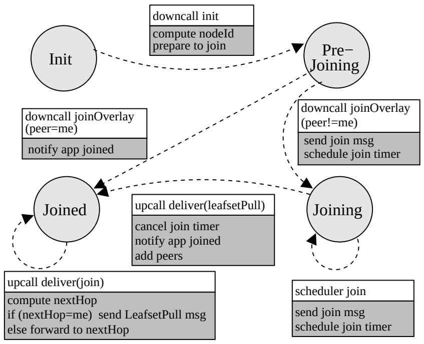

Figure 4. State machine model for Bamboo. States are shown in light gray; event-transitions are represented by arrows with names in white and actions in dark gray.

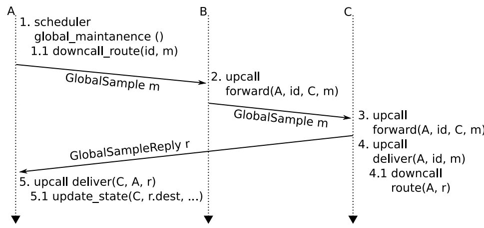

Figure 5. Message diagram for global sampling. In response to a scheduled timer, Node A routes a $GlobalSample$ message to an identifier id by sending it to node B, which is forwarded to its owner, node C. Node C responds with a $GlobalSampleReply$ message informing node A about node C, potentially causing a routing table update.

```
detect {
  guard    = (range != pre(range));
  error    = notifyNewRange;
} // local detection
detect {
  guard    = (state == joined);
  nodes    = myleafset;
  send     = { message =
                 LeafsetPush(myhash, myleafset);
               period = 5sec; }
  receive  = { message = LeafsetPull;
               period = 5min; }
  error    = leafFailed;
} // distributed detection (across myleafset)
```

Figure 6. Local and Distributed inconsistency and failure detection in Mace.

state machine for the BAMBOO object illustrated in Figure 4. The figure shows what happens when BAMBOO receives events such as application join requests, network messages, or timers causing reattempted joins. The system begins in the `init` state, and transitions to the `preJoining` state upon receiving a downcall event `init` from the application. When it subsequently receives the downcall event `joinOverlay`, it transitions to the `joining` state if it is not its own `peer` (captured via a predicate guarding the event), or to the `joined` state otherwise, in either case sending appropriate notification events and scheduling timers. In the `joining` state it periodically sends `join` messages to other nodes requesting to join the system, and finally, when it receives a `deliver(leafsetPull)` message from another node, it moves into the `joined` state. The rest of a node's life is spent in the `joined` state, where it periodically globally samples the other nodes to improve its local routing information.

Sequence diagrams are an informal technique programmers use to reason about low level interactions, such as those taking place while in the `joined` state. Figure 5 shows a sequence diagram depicting the interaction between nodes performing global sampling to improve routing tables. The periodically scheduled `global_maintenance` timer fires on node `A` causing it to select a random routing identifier `id`, and to then send a `GlobalSample` message to the node `B`, which is the next hop along the route. This message gets (transitively) forwarded by `B` until it reaches `C`, which actually owns the identifer `id`. `C` then sends a `GlobalSampleReply` message back to `A`, which, upon receiving the reply, may update its routing information.

Once the programmer has worked out the details of the protocol using the sequence diagram, it is straightforward to code in Mace using transitions and events. Figure 3 shows (in order) how the events corresponding to i) the firing of the `global_maintenance` timer at `A`, ii) the forwarding of the `GlobalSample` message to `B` and `C`, iii) the final delivery message to the destination `C`, and iv) the delivery of the `GlobalSampleReply` back to `A`, are implemented as transitions in the BAMBOO service object. The bodies of the respective transitions implement the actions taken upon receiving the corresponding events shown in Figure 5.

3.3 Failures

Mace's use of service objects and events greatly simplifies the task of detecting, notifying, and handling failures and inconsistencies. While layering is essential for building complex services, the information hiding endemic to layered systems often makes it difficult to deliver the best performance or the most agile fault handling. For example, when a DHT application's socket breaks due to a node failure, the TCP transport layer could attempt to mask the error. However, doing so may prevent BAMBOO from being able to route around the failed node, leading to degraded performance or incorrect message delivery. Rather, the TCP transport layer must propagate the error to BAMBOO so that it can update its routing table and leafset. BAMBOO, in turn, further propagates the error to the DHT application, so that it can redistribute the keys stored on the failed node. Mace provides clean mechanisms for layering network services while also making it easy to deliver error notifications automatically from one layer to another when required for performance or fault tolerance.

Mace employs upcalls to signal higher layers of potential performance and correctness issues. It may be possible to correctly handle an issue entirely at a lower layer, but with suboptimal performance. If this is acceptable to upper layers, then they can simply ignore the corresponding upcall. However, for best performance it may be necessary to register handlers for such upcalls. While such cross-layer communication usually obfuscates code and eliminates many of the benefits of layering, we leverage our event-based structure to cleanly separate the *notification* and *recovery* code from the rest of the system that executes in the non-exceptional case.

Mace addresses the remaining challenge of providing programmers with a succinct but flexible mechanism for *detecting* both failures and inconsistencies through the use of *aspects*. Aspects provide a unified way to maintain consistent state, regardless of whether the state needs to be updated in response to an expected protocol event, such as a node arrival, or an unexpected event, such as a failure. Mace aspects check for two types of inconsistency/failure detection: those that involve purely local state and those that involve multiple nodes.

Local Failure Detection. Failures occurring in distributed systems can be characterized via inconsistencies in the values of state variables. A *local* failure occurs when the values of state variables at a single node are inconsistent. For example, in our DHT application built on top of BAMBOO, the data that each node is responsible for depends on the key space specified by the `range` variable. In other words, the views of the range of the DHT layer and the BAMBOO layer must be synchronized, and if they are not, recovery action must be taken so that the DHT relocates the data according to the new range, potentially involving communication with remote system nodes.

Such failures can be specified using a predicate that characterizes the inconsistent values, *i.e.*, which becomes true when the values of the variables are inconsistent. Thus, such failures can be locally detected by monitoring the predicate, and firing an event when the predicate becomes true. In Mace, the programmer specifies how failures should be detected and how to react to the failure using a *detection aspect*. A failure occurs when this predicate is true, which fires an error event and notifies the upper layers of the inconsistency.

Consider the example in the top of Figure 6, showing a local detection aspect that specifies that an inconsistency failure occurs when the value of the variable `range` changes (the `pre()` version refers to the value of `range` before the last transition). When the change occurs, Mace sends a `notifyNewRange` event to the upper DHT layer indicating that its portion of the key space has changed and prompting it to reorganize stored data appropriately. This aspect will correctly react to *all* events that change the range, whether it be the arrival of a new peer adhering to the BAMBOO protocol or an unexpected peer failure.

The local detection aspect checks the predicate only at transition boundaries, avoiding notification of state that may become temporarily inconsistent in the middle of a transition. Thus, aspects and events provide a clean way to separate the failure detection, notification, and handling from the rest of the "common-case" code. We implement detection by keeping a shadow copy of monitored state variables, checking and updating them after each transition. In ad-hoc implementations, built without language support, the programmer would have to manually insert the check and notification each time the variables might be modified. In addition to greatly reducing readability, this task is error-prone, especially as the code evolves or is maintained by multiple programmers.

Distributed Failure Detection. A *distributed* failure occurs when the values across two or more nodes are inconsistent. For example, in BAMBOO, each node maintains the set of its immediate peers in the state variable `myleafset`. Each such peer, in turn, must include the node in its own set of known peers. A distributed failure occurs if some element of a node's leafset does not include the node in its own set of peers. As a node cannot directly access the other nodes' internal state, the only way to determine the presence of such a failure is to actively exchange information across nodes, checking that the received information is consistent, and if so, returning messages acknowledging consistency. If the originating node receives the acknowledgment before a timeout occurs, it confirms that no failure has occurred.

These failures can also be captured via predicates over the state variables of multiple nodes, and the Mace compiler automatically generates the periodic probe and acknowledgment messages. However, it is profitable to let the programmer control how the probing happens to avoid flooding the network with messages pertaining to failure detection.

The programmer specifies how to detect and react to distributed failures in Mace using the same *detection aspect* as for local failures, but now defines more elements for the aspect, as shown at the bottom of Figure 6. The guard specifies the conditions for performing a probe. The `nodes` are the set of nodes monitored by the aspect. In this example, it is the nodes stored in the set `myleafset`. The `send` field indicates how the probes are sent to the elements of `nodes`. Here, the node sends `LeafsetPush` messages with the current value of `myhash` and `myleafset` state variables to the elements of `myleafset` sequentially, once every 5 seconds. The `receive` field indicates the response expected from the other nodes, together with a timeout before which the response must arrive. Here, it stipulates that the node must receive a `LeafsetPull` message from each of the other nodes in `myleafset` before a timeout period of 5 minutes elapses.

Mace generates extra state and the code needed to keep track of the last time it has heard from each monitored node. It sets a timer to fire sometime after it expects to receive an acknowledgment of a particular remote configuration. If the timeout occurs and the guard is true, then Mace calls the event specified in the `error` field, notifying upper layers of the failure. As in the case of the local failures, the transition corresponding to the error event corresponds to the code that implements the recovery mechanism. Likewise, Mace simplifies the detection of distributed failures by separating the detection code into an aspect and automating the process of sending the probe messages and detecting timeouts.

3.4 Analysis

By using objects and events to preserve the high-level structure of the distributed system, Mace can automate a variety of post-development analyses that find performance or correctness problems.

Execution Logging and Debugging. Mace uses aspects to generate debugging and logging code without cluttering the service specification. Mace exploits the preserved structure to enable different levels of automatic logging. First, with event-level logging, the generated program logs the beginning and end of each high-level event. This captures the order and timing of events at each node. With state-level logging, every time a transition finishes, the generated program logs the node's complete state, which indicates the change caused by the transition. Finally, with message-level logging, the generated program additionally logs the content and transmission time for each message sent from or received by the node.

We have used the automatically generated logs to implement MDB, a replay debugger for Mace distributed applications. MDB collects all individual node log files centrally and allows the developer to single step, forward and backward, through the execution of individual nodes. The developer may move from node to node, inspecting global system state, in a manner similar to traditional single process debuggers. Our ability to swap in a simulated messaging layer further allows the developer to explore alternate execution paths, diverging from some given point in a real execution.

Causal-Paths. Mace provides a more advanced form of logging that aggregates execution events distributed across multiple nodes into a set of causal paths. Each path starts at a given node, with a particular seed event, and contains the sequence of all events that are causally, transitively related to the seed event. For example, if the seed event is a request generated by a particular node, then the causal path includes the sequence of messages (and resulting events and transitions) that span the different nodes until the response returns to the requester.

To obtain such causal paths in a Mace application, the programmer specifies the seed event where the path begins and the event that ends the path. Mace tags all the relevant, causally related activity that occurs between the seed and the end (*i.e.*, all events, transitions, messages sent and received) with a dynamically generated path identifier and generates logs such that events distributed across multiple nodes can be collected using their shared path identifier. As a result, Mace enables logging at a semantic-level and allows programmers to understand and analyze the behavior of the system at a high level. In addition, previous work [29] describes how the causal-path logging done by Mace can be *automatically* mined to find and fix performance anomalies, by comparing the causal paths resulting from actual executions with programmer-specified high-level *expectations*.

While this earlier work on the benefits of causal paths to performance debugging is independent of Mace, it requires significant manual logging in standard, unstructured C++ applications. We have found that the more than 90% of the logging required for causal path analysis can be automatically inserted by the Mace compiler, significantly lowering the barrier for leveraging the benefits of such performance debugging tools.

Model Checking. A high-level model of a distributed system enables exhaustive analyses like model checking to find subtle bugs in either the protocol or implementation of a distributed system. Mace allows developers to use the same analysis to find subtle errors in the actual implementation of the system by making it easy to systematically explore the space of executions of the implementation. We have built MaceMC [17], a model checker targeting liveness violations in Mace. While our techniques for finding liveness violations in real systems implementations are general (these techniques are the focus of [17]), here we describe the benefits of Mace language structure in integrating a software model checker:

- Mace's service layering mechanism simplifies integrating a simulation engine by replacing the services implementing the actual network with a simulated network of queues holding the messages between nodes. In ad hoc implementations, messaging functionality may be spread throughout the code making it difficult to plug in the simulated messaging layer necessary for model checking.
- Mace's state-event semantics ensures that a node's state changes by processing a single event *atomically*, via a single transition. Thus, the model checker need only consider event interleavings across individual events at participating nodes, rather than, for instance, exploring all interleavings at a much finer granularity or forcing the developer to manually identify appropriate transition points.
- Mace's state-event semantics allow us to deterministically *replay* an execution (once the sequence of events is fixed) either for the purpose of demonstrating a buggy execution, or to perform a random simulation from a previously visited state. This allows MaceMC to exhaustively search the states up to a certain depth and to then perform *deep random walks* from the boundary states to look for liveness violations. We have found that without such random walks it is impossible to distinguish between actual (permanent) liveness violations and temporary divergences from desired high-level system properties.

After applying MaceMC to 5 significant systems implementations (a subset of those mentioned in Section 4), we were able to find 50 subtle protocol bugs. Most of these bugs were present in systems that had already been hardened through live Internet deployment and manual debugging.

4. Experiences

In this section, we outline some of our experiences developing distributed applications with Mace. Mace itself is implemented as a source-to-source compiler in Perl using a recursive descent parsing module. The Mace compiler emits C++ code, which is then compiled using any C++ compiler such as g++. We have implemented over nine substantial distributed systems in Mace, many of which we have run across the Internet, including on testbeds such as PlanetLab [28]. In addition to the BAMBOO implementation discussed here, we have also implemented the systems shown in Figure 7. These systems include CHORD [35], PASTRY [32], SCRIBE [33], SPLITSTREAM [7] (from the FreePastry [1] distribution), BULLETPRIME [20] (from the MACEDON [31] distribution), OVERCAST [15] (not available to us for line counting), and VIVALDI [9]. Excepting BULLETPRIME (which was written in the MACEDON language), each of these services were originally developed in unstructured C++ or Java.

The Mace compiler eliminates many tedious tasks that must otherwise be hand-implemented to achieve high performance, such as message serialization and event dispatch, and correspondingly drastically reduces the implementation size. A Mace service object implementation contains a block for specifying message types (essentially a `struct` with optional default values), for each of which the compiler generates a class containing optimized methods to serialize and deserialize the message to and from a byte string that can be sent across the network. The Mace compiler also generates methods to automatically perform event sequencing and dispatch. The generated code selects the next pending event, performs locking to prevent preemption, evaluates any guard tests for the transition, executes the appropriate method implementing the event handler (assuming the guards succeeded), tests any aspect predicates that might have been updated by the transition, and finally releases the acquired locks. Overall, we find that the structure imposed by Mace greatly simplifies the implementation by allowing the programmer to focus only on the essential elements, without compromising performance or reliability.

4.1 Performance Evaluation

To evaluate the performance of Mace systems, we compare our BAMBOO implementation in Mace with its well-tested counterpart [30]. To distinguish the two versions, for this section we will refer to our implementation as Mace-Bamboo. We chose BAMBOO because of its excellent performance, detailed published performance evaluation, and its publicly available and well documented code base. Bamboo is a highly optimized Java implementation of a distributed hash table, based originally on Pastry [32]. We compare behavior of node lookups under churn. Lookups operate by forwarding a message using increasing prefix matching to nodes whose identifiers are progressively closer to the key. Bamboo explores the limitations of previous protocols in providing consistent routing in the presence of node churn, and proposes several modifications to Pastry to allow it to deliver high consistency and low latency even when nodes are entering and leaving the system at a high rate.

Consistency is a measure that captures whether different nodes routing to the same identifier will reach the same destination. This is the most important requirement for correct performance of applications using a DHT, since they rely on being able to share data by using the same identifier to store and retrieve values. Our exercise of re-implementing Bamboo serves to show the simplicity of implementing distributed systems in Mace and our ability to generate robust, efficient, and high performance code. Two experienced Mace developers implemented the primary Bamboo algorithms in twelve hours (excluding the reliable UDP transport), starting from an existing Mace Pastry implementation.

To compare against published Bamboo experimental results (we attempted to reproduce the published results but could never achieve them, most likely due to having fewer machines), we prepare a framework that matches, to the best of our ability, the original experimental conditions. The experiment consists of 1000 Bamboo nodes organized into groups of 10 performing simultaneous lookups of random keys. A lookup result is considered consistent if a majority of the 10 nodes return the same result. Each group of 10 nodes performs lookups according to a Poisson process with an average inter-lookup delay of 1 second. For the runs, we vary the median churn rate also according to a Poisson process, ranging from on average 8 deaths per second to 1 death per second.

We run 1000 Bamboo instances on 16 physical machines (the published BAMBOO results used 40 machines), using the ModelNet [36] network emulator with a single FreeBSD core. Each of the physical machines is a dual Xeon 2.8Mhz processor with 2GB of RAM. During the runs, load averages ranged from 0.5 to 1.5. The emulated topology consists of an INET network with 10,000 nodes, 9,000 of them routers. Client bandwidths on the topologies ranged from 2-8Mbps. To start the experiment, nodes were staggered, starting one on each machine each second for a minute. The churn and lookup schedules began as soon as all nodes were live. This experimental setup differs from the published Bamboo exper-

System	Mace	Distribution
Bamboo	500	1800
BulletPrime	1000	2800
Chord	250	3400
Overcast	450	NA
Pastry	600	3600
Scribe	300	500
SplitStream	200	331
Vivaldi	100	250

Figure 7. Lines of code measured in semicolons for various systems implemented in Mace and other distributions.

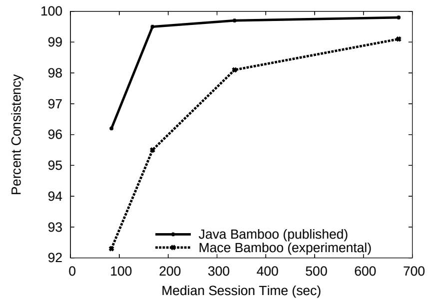

Figure 8. Percentage of lookups that return a consistent result.

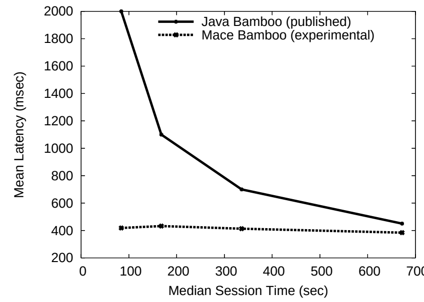

Figure 9. Mean latency for lookups that return consistent results.

iments in that the stagger-start is at a much faster rate, that we do not wait for the network to settle after starting all nodes, that we run with 1/3 the number of machines, and that our request load is 10 times higher.

Figure 8 shows the consistency numbers for the Java-Bamboo and Mace-Bamboo. The published consistency values demonstrate near-perfect consistency at all churn levels. While still above 92% consistent, Mace-Bamboo nodes are slightly less consistent than their counterpart, though they track its performance closely. However, as shown in Figure 9, the Mace-Bamboo latency outperforms Java-Bamboo at each of these churn levels, and by a factor of 5 at high churn levels.

4.2 Undergraduate Course

To aid in the evaluation and development of Mace, we have used it in two undergraduate networking courses, and it has also been used in several graduate course projects. During the Spring Quarter 2005 and the Spring Quarter 2006, students in advanced undergraduate networking classes were "asked" to program in Mace for a class project. None of the students enrolled in the class had been exposed to Mace previously. The project involved implementing a peer-to-peer file sharing program loosely based on the popular FastTrack protocol. The protocol includes a number of distributed concepts such as flood-based searching, distributed election, and random network walks. To prepare for the project, students were given a one-hour introduction to Mace, a list of protocol messages (to support inter-operation), and a skeleton template for a basic Mace service. 90% of the students successfully completed the project and a majority expressed a preference for programming in Mace relative to Java or C++.

5. Related Work

Mace is closely connected to a large body of work in the area of languages, libraries and toolkits for building concurrent systems. We focus our attention on those specific to distributed systems.

We build upon our earlier MACEDON [31] work—a domain specific language for fair comparisons of overlay systems. MACEDON also represents systems as I/O automata, but does not consider how compiler extensions that restrict specifications can support model checking, high performance, debugging, etc. Whereas MACEDON focused on building prototype lab-experiments, we designed Mace as a practical, real-world environment for developing deployable high-performance, reliable applications.

The *state-event-transition* model Mace is founded on is closely related to other event-driven languages and libraries. NesC [12] is a language for building sensor networks with limited resources requiring static memory allocation. Broadly speaking, several researchers have investigated providing language support for building concurrent systems out of interacting components, such as CLICK [19] for building routers from modules and the FLUX OSKIT [11] for building Operating Systems. These approaches, however, target concurrent systems executing within one physical machine and not distributed systems scattered across a network.

P2 [22] is a declarative, logic-programming based language for rapidly prototyping overlay networks specifying data-flow between nodes using logical rules. While P2 specifications are substantially more succinct than those in Mace, the corresponding specification is not as natural to programmers and sacrifices performance. Also, while Mace is well-suited for building overlays, its applicability is broader. There is a line of work in the functional programming community for advanced, type safe languages for distributed computation [34]. At the moment these languages are somewhat experimental, emphasizing fully understanding the semantics of high-level constructs for distributed programming and their interplay with the type system rather than enabling the rapid deployment of robust, high-performance distributed systems.

Several libraries and toolkits also support many of the common primitives required for building distributed systems. LIBASYNC [25] uses a single-threaded event driven model that makes extensive use of callbacks. LIBASYNC also provides some compiler support for dealing with Remote Procedure Calls and for generating some of the serialization code for messaging. Another instance, SEDA [37], provides an architecture for event-based systems. Both of these systems focus on simplifying the implementation of event-driven code, rather than a structure for distributed systems.

Mace uses ideas proposed by Aspect-Oriented Programming (AOP) [16]. One of the first examples of AOP was a domain-specific language for writing distributed software [23]. A primary contributions of Mace is identifying the different *concerns* that comprise a distributed system—*e.g.* the messages, events, transitions, failures, and logging—and designing a language that enables programmers to think about these in isolation. The Mace compiler seamlessly puts each of these together to create an efficient implementation of the system. An immediate payoff of this separation is the ease with which a programmer can log and monitor entire event flows without cluttering the code with print statements.

There are several high-level languages for describing network protocols, rather than entire distributed systems. Some of these—*e.g.* LOTOS [5], ESTELLE [6]—are intended largely to formally specify protocols using finite state machines that communicate by passing messages. PROMELA [14] and TLA [21] are two more general languages which can be used to model concurrent systems. Instead of producing executable systems, they compile the description into large finite state machines to exhaustively analyze for errors. RTAG [3] based on grammars and PROLAC [18] based on an object-oriented model are two examples of protocol description languages that actually compile the description into executable code. Mace combines the benefits of both by structuring the description of the system such that the subsequent compiled implementation is amenable to exhaustive analysis.

6. Conclusions

In this paper, we argued for the benefits of language support to construct robust, high-performance distributed systems. The principle challenge in this environment is resolving tensions between the tasks of developing a clean layering system, handling concurrency and failures, and preserving enough structure to enable automated performance and correctness analyses. The key insight behind Mace is that objects, events, and aspects can be seamlessly combined to simultaneously address the intertwined challenges.

Mace's language structure and restrictions enable a number of important features that are otherwise difficult or impossible to express in existing languages: language support for failure detection, causal path performance and correctness debugging, and model checking unmodified Mace code. We have employed Mace to build ten significant distributed applications, which have been successfully deployed over the Internet. Others are using Mace to support their own independent research and development. Using automated debugging tools that exploit the Mace structure to find and fix problems, exploiting the flexible architecture to reuse optimized subsystems across applications, and leveraging the uniform and efficient event-driven concurrency model, Mace system specifications were about a factor of five smaller than original versions in Java and C++, while delivering better performance and reliability.

References

[1] Freepastry: an open-source implementation of pastry intended for deployment in the internet. `http://freepastry.rice.edu`, 2006.

[2] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP* (2003).

[3] ANDERSON, D. P. Automated protocol implementation with RTAG. *IEEE Trans. Software Eng. 14*, 3 (1988), 291–300.

[4] BARHAM, P. T., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: Online modelling and performance-aware systems. In *Proc. HotOS* (2003).

[5] BOLOGNESI, T., AND BRINKSMA, E. Introduction to the ISO specification language LOTOS. *Computer Networks 14* (1987).

[6] BUDKOWSKI, S., AND DEMBINSKI, P. An introduction to Estelle: A specification language for distributed systems. *Computer Networks 14* (1987), 3–23.

[7] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., AND SINGH, A. SplitStream: High-bandwidth content distribution in cooperative environments. In *Proc. SOSP* (2003).

[8] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. DSN* (2002).

[9] DABEK, F., COX, R., KAASHOEK, F., AND MORRIS, R. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference* (Portland, Oregon, 2004).

[10] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM Press, pp. 202–215.

[11] FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., AND SHIVERS, O. The Flux OSKit: A substrate for kernel and language research. In *Proc. SOSP* (1997).

[12] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: A holistic approach to networked embedded systems. In *Proc. PLDI* (2003).

[13] GODEFROID, P. Model checking for programming languages using VeriSoft. In *Proc. POPL* (1997).

[14] HOLZMANN, G. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[15] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND JAMES W. O'TOOLE, J. Overcast: Reliable multicasting with an overlay network. In *Proc. OSDI* (2000).

[16] KICZALES, G. Aspect-oriented programming. *ACM Computing Surveys 28*, 4es (1996).

[17] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *NSDI* (2007).

[18] KOHLER, E., KAASHOEK, M. F., AND MONTGOMERY, D. R. A readable TCP in the Prolac protocol language. In *Proc. SIGCOMM* (1999).

[19] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM TOCS 18*, 3 (2000), 263–297.

[20] KOSTIĆ, D., BRAUD, R., KILLIAN, C., VANDEKIEFT, E., ANDERSON, J. W., SNOEREN, A. C., AND VAHDAT, A. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX Tech* (Anaheim, CA, Apr. 2005).

[21] LAMPORT, L. *Specifying Systems: the Tla+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[22] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In *Proc. SOSP* (2005).

[23] LOPES, C. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1996.

[24] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[25] MAZIERES, D. A toolkit for user-level file systems. In *Proc. USENIX Tech* (2001).

[26] MILNER, R. *Communication and Concurrency*. Prentice-Hall, 1989.

[27] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: a read/write peer-to-peer file system. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation* (New York, NY, USA, 2002), ACM Press, pp. 31–44.

[28] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets–I)* (Princeton, New Jersey, 2002).

[29] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[30] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a dht, 2004.

[31] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIĆ, D., AND VAHDAT, A. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc. NSDI* (2004).

[32] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware 2001* (2001).

[33] ROWSTRON, A., KERMARREC, A.-M., CASTRO, M., AND DRUSCHEL, P. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. Third International Workshop on NGC* (2001).

[34] SEWELL, P., LEIFER, J. J., WANSBROUGH, K., NARDELLI, F. Z., ALLEN-WILLIAMS, M., HABOUZIT, P., AND VAFEIADIS, V. Acute: high-level programming language design for distributed computation. In *Proc. ICFP* (2005).

[35] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer to peer lookup service for internet applications. In *Proc. SIGCOMM* (2001).

[36] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIĆ, D., CHASE, J., AND BECKER, D. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. OSDI* (2002).

[37] WELSH, M., CULLER, D., AND BREWER, E. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. SOSP* (2001).