Building Reliable Distributed Systems With P

Ankush Desai
Ethan Jackson
Amar Phanishayee
Shaz Qadeer
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-198
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-198.html

September 15, 2015

Acknowledgement

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP)
a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Building Reliable Distributed Systems With P

Ankush Desai[‡†], Ethan Jackson[†], Amar Phanishayee[†], Shaz Qadeer[†], Sajit Seshia[‡]
[†]*Microsoft Research,* [‡]*UC Berkeley*

Abstract

Fault-tolerant distributed systems are difficult to get right because they must deal with concurrency and failures. Despite decades of research, current approaches for a more rigorous way of building robust distributed systems are unsatisfactory. In this paper, we present P, a new approach that makes it easier to build, specify, and test distributed systems. Our programming framework provides two important features. First, we provide a high-level language for formally specifying implementations, abstractions, and specifications of protocols. The P compiler automatically generates efficient C code from the input program. Second, we provide a tool for compositional systematic testing of a P program. Together, these attributes have the power to generate and reproduce within minutes, executions that could take months or even years to manifest in a live distributed system.

1 Introduction

Fault-tolerant distributed systems must reliably handle concurrency and failures. Programming with concurrency and failures is challenging because of the need to reason about numerous program control paths. These control paths result from two sources of nondeterminism—interleaving of event handlers and unexpected failures. Not surprisingly, programmers find it difficult to reason about the correctness of their implementations. Even worse, it is extremely difficult to test distributed systems; unlike sequential programs whose execution can be controlled via the input, controlling the execution of a distributed program requires fine-grained control over the timing of the execution of event handlers and fault injection. In the absence of such control, most control paths remain untested and serious bugs lie dormant for months or even years subsequent to deployment. Finally, bugs that occur during testing or after deployment tend to be Heisenbugs; they are notoriously difficult to reproduce because their manifestation requires timing requirements that might not hold from one execution to another. These problems are well-known and have been highlighted by creators of large-scale distributed systems [8]. Unfortunately, despite decades of research in verification techniques oriented towards distributed systems, the practice of programming such systems "in-the-wild" has not changed.

Existing validation methods for distributed systems fall into two broad categories. On one hand, we have systems such as TLA+ [36], a mathematical logic for specifying distributed protocols. Although TLA+ supports rich specification and validation via model checking, the realm of logical specifications is far removed from the imperative languages preferred by programmers of distributed systems. Consequently, there is no formal connection between the specification being checked and the code being executed. On the other hand, we have systems such as MODIST [35], a model checker that operates directly on the implementation of a distributed system. This approach validates real executions but lacks scalability in the face of the daunting complexity of realistic distributed systems.

In this paper, we present P, a scalable approach to implement, specify, and test distributed systems. P rests on two important pillars. First, we provide a high-level language for describing the implementation of a distributed system. This language has precise operational semantics and is based on the computation model of asynchronously-communicating state machines, where each state machine is equipped with an input buffer containing messages to be processed. We provide a compiler that automatically generates efficient C code from the input program and a runtime to execute the compiled program on a cluster of computing nodes. Since the management of state machines and event buffers is handled automatically by the runtime, P programs tend to resemble high-level protocol specifications that are significantly more compact than corresponding C implementations. The conformance of the semantics of P programs to executing code ensures that programmers do not have to worry about the high-level protocol specification diverging from the low-level implementation.

Second, we provide a tool for compositional systematic testing of a formally-specified protocol. Systematic testing means that protocol executions are generated automatically by enumerating all sources of non-determinism in the formal specification. Compositional testing means that the executions of a protocol are tested in isolation by abstracting the environment in which the protocol

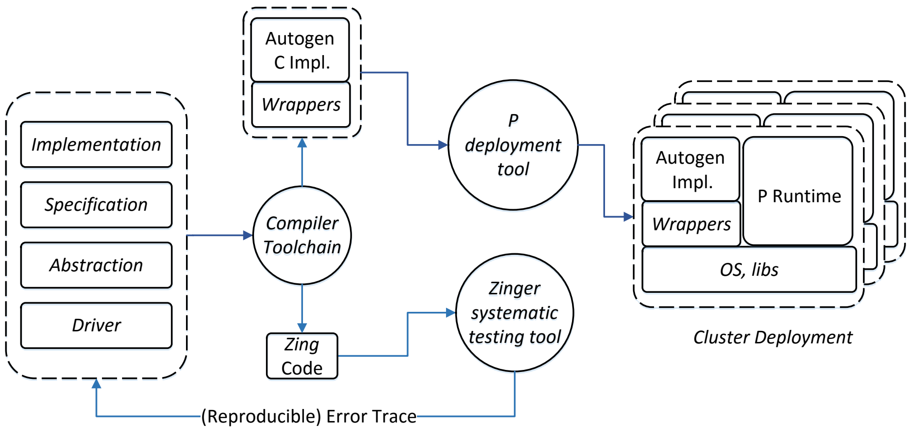

Figure 1: Overview of P

executes. Large distributed systems are typically composed of many protocols interacting with each other. For example, fault-tolerant distributed transaction commit [14], uses State Machine Replication (SMR) to make two-phase commit fault-tolerant; protocols for SMR, like Multi-Paxos [22] and Chain Replication [32], in turn use other protocols like leader election, failure detectors, and network channels. Thus, the environment of a protocol often consists of other protocols. We introduce primitives in our programming language for writing protocol abstractions, which are exploited by our tool to test each protocol separately. Compositional reasoning boosts the scalability of our testing framework and provides much greater behavior coverage per unit of computation devoted to testing. Compositional systematic testing has the power to generate and reproduce within minutes, executions that could take months or even years to manifest in a live distributed system.

We use P to implement two distributed services: (i) distributed atomic commit of updates to decentralized, partitioned data using two-phase commit, and (ii) distributed data structures such as hashtables and lists. These services use State Machine Replication (SMR) protocols, like Multi-Paxos and Chain Replication written in P, for fault-tolerance.

We summarize the contributions of the paper:

- We have designed the P language for succinctly and precisely describing the implementation of a distributed system. Our language compiler and runtime automatically map the semantics of P programs to code executing on a cluster.
- We introduce language primitives in P for expressing specifications and abstractions of protocols in a distributed system. We exploit these specifications to test each protocol in the system separately, thereby tremendously increasing the scalability of our testing framework.
- We present an empirical evaluation of systematic testing and runtime performance in P using two distributed services that combine ten different protocols, thus demonstrating that our framework can validate complex real-world distributed services.

2 P Overview

Figure 1 provides an pictorial overview of the P architecture. There are three main building blocks of the P framework—a *programming language* for implementing and specifying a distributed system, a *testing* tool for efficiently exploring the nondeterminism in a P program, and a *runtime library* for efficiently executing the program.

P program: A P program consists of four main components, *implementation*, *specification*, *abstraction* and the *test-driver*. The *implementation* block contains the protocols implementing the distributed system. The computational model underlying a P program is state machines communicating via messages, an approach commonly used in building networked and distributed systems [6, 20, 21, 29, 34]. Each state machine has an input queue, event handlers, and machine-local store for a collection of variables. The state machines run concurrently with each other, each executing an event handling loop that dequeues a message from the input queue, examines the local store, and can execute a sequence of operations. Each operation either updates the local store, sends messages to other machines, or creates new machines. In P, a send operation is non-blocking; the message is simply enqueued into the input queue of the target machine.

The *specification* block captures the correctness properties of the implementation. The specifications of the system are implemented in the form of *monitors*. Each monitor is a state machine that maintains necessary state for asserting a temporal safety or liveness requirement [3]. The implementation is instrumented to send relevant events to each monitor enabling it to check if the system satisfies the specification.

Testing a large distributed system is not scalable as the number of possible executions is extremely large. Instead, P allows the programmer to test each protocol in the system separately, a methodology we call *compositional testing*. It is not possible to test a protocol in isolation because it interacts with other protocols. These interactions must be modeled accurately to ensure no false positives during testing and abstractly to ensure that testing a protocol in isolation is simpler than testing the entire system. P solves this problem by allowing the programmer to specify protocol abstractions also as state machines. These abstract machines, known as *model* machines constitute the *abstraction* block in a P program. While testing a particular protocol, the P testing tool automatically replaces any protocols in its environment with the appropriate abstractions.

The *test-driver* block is key to scalable systematic testing of a P program. Each protocol has a separate test-driver to capture the set of executions that must be explored during systematic testing of that protocol. The driver uses P features to compactly specify a large set of executions to be tested. These executions are then automatically generated by the systematic testing tool in P.

Systematic testing: Our tool for systematic testing of P programs is implemented in two parts. The P compiler generates a translation of the program into the Zing modeling language [4]. Next, the Zinger tool takes as input the generated Zing model and systematically enumerates executions resulting from scheduling and explicit nondeterministic choices. Together, these two steps create a single-process interpreter and explorer for the nondeterministic semantics of a P program. A programmer typically spends the initial part of protocol development in the iterative edit-compile-test-debug loop enabled by our systematic testing tool (see the feedback loop in Figure 1). The feedback from the tool is an error trace that picks a particular sequence of nondeterministic choices leading to the error. Since all nondeterministic choices are explicitly indicated by the tool, the process of debugging becomes considerably simpler that the current practice of correlating logs created at different machines in a live execution.

Execution: The P compiler also generates C code that is compiled by a standard C compiler and linked against the P runtime to generate the executable of the distributed service. The application is deployed on a collection of machines in a cluster by the P deployment tool. Runtime ensures that the behavior of a P program matches the semantics validated by the systematic testing.

Listing 1 Timer.

```
// events from client to timer
event START;
event CANCEL;
// events from timer to client,
// each accompanied by a machine id
event TIMEOUT: machine;
event CANCEL_SUCCESS: machine;
event CANCEL_FAILURE: machine;
// local event for control transfer within timer
event Unit;
model Timer {
  var client: machine;
  start state Init {
    entry {
      client = payload as machine;
      raise Unit;  // goto handler of Unit
    }
    on Unit goto WaitForReq;
  }
  state WaitForReq {
    on CANCEL goto WaitForReq with {
      send client, CANCEL_FAILURE, this;
    };
    on START goto WaitForCancel;
  }
  state WaitForCancel {
    on START do { };
    on CANCEL goto WaitForReq with {
      if ($) {
        send client, CANCEL_SUCCESS, this;
      } else {
        send client, CANCEL_FAILURE, this;
        send client, TIMEOUT, this;
      }
    };
    // if there is no START or CANCEL event
    on null goto WaitForReq with {
      send client, TIMEOUT, this;
    };
  }
}
```

3 Modeling asynchronous interaction and failures

The goal of P is to enable development of distributed systems in a manner that facilitates systematic testing of unexpected executions that result from asynchronous interaction and failures. In this section, we show that suitable combination of two primitives of the P programming language, concurrently-executing state machines and nondeterministic choice, can precisely and succinctly model these features of distributed systems.

Asynchronous interaction is modeled naturally in P by implementing a distributed system as a collection of concurrently-executing state machines that communicate using buffered non-blocking sends. To systematically test distributed systems, we need to capture interactions with external libraries, such as OS timers and file systems, not implemented in P. In addition to describing the implementation of a distributed system, state ma-

Listing 2 Unreliable send.

```
machine Sender {
  ...
  model fun DoRPC(val:any, seqNum:int): bool {
    if ($) return false;
    send dst, RECV,
         (from=this, val=val, seqNum=seqNum);
    return $;
  }
  ...
}
```

chines also enable us to write abstractions (models) of external libraries. We capture such components in P either as *model* state machines (asynchronous interaction) or as *model* functions (synchronous interaction). As an added benefit, abstractions also model P implementations and enable compositional testing. When testing a protocol in isolation, we replace the other protocol implementations it interacts with by an appropriate abstraction to reduce the search space that must be explored by the testing tool.

To illustrate how asynchronous interaction is modeled in P, we consider a client program using a timer, a basic OS primitive required for distributed programming. To start a timer the client makes an asynchronous call to `StartTimer(10, Callback)`. This call tells the OS to invoke the function `Callback` after 10 ms, but the `Callback` can happen anytime because of asynchrony. Subsequently, the client program may cancel the timer with a system call `CancelTimer()`. Timer cancellation may not succeed if the timer has expired or is about to expire. The timer library guarantees that if timer cancellation succeeds, then `Callback` is never invoked.

Since `StartTimer` is an asynchronous callback, we must model the client and the timer as concurrently executing processes. Further, to avoid reasoning about real time and the detailed internal state of the timer library, we must introduce nondeterministic choice while modeling the timer behavior from the client's perspective. Thus, we allow `Callback` to start executing any time after `StartTimer` is invoked.

Listing 1 shows a timer model machine in P to illustrates how we can precisely express the behavior described above. This example also introduce key features of the P language useful for understanding other examples described later in the paper.

A client machine with a local variable `timer` may create an instance of the `Timer` state machine via a constructor call `timer = new Timer(this)`, passing its own id as an argument so that the timer can later send events to the client. The code for `Timer` state machine contains variables and states. For each state, event handlers may be specified for specific events; the appropriate handler is executed upon dequeueing an event from the input queue of the machine. The variables are local to

Listing 3 Fault injector.

```
model FaultInjector {
  var nodes: seq[machine];
  var i: int;
  start state Init {
    entry {
      nodes = payload as seq[machine];
      raise Unit;
    }
    on Unit goto Fail;
  }
  state Fail {
    entry {
      if (sizeof(nodes) == 0)
        return;
      i = 0;
      while (i < sizeof(nodes)) {
        if ($) {
          send nodes[i], halt;
          nodes -= i;
          raise Unit;
        } else {
          i = i + 1;
        }
      }
    }
    on Unit goto Fail;
  }
}
```

an instance of `Timer` but global to all event handlers across all states.

A freshly-created `Timer` machine starts in the state `Init`; upon entering `Init` the entry action is executed to initialize the `client` variable. The assignment performing this initialization uses the keyword `payload` to access the argument to the constructor. The `payload` is allowed to be of any type; in particular, it could be a tuple to send multiple arguments. This keyword is also used to access the data value attached to an event sent from one machine to another.

In addition to `Init`, `Timer` has two other states—`WaitForReq` and `WaitForCancel`. In state `WaitForCancel`, nondeterminism (`$`) is used in the handler for `CANCEL` event to model success or failure of the cancellation request. In case of success, timer reponds with `CANCEL_SUCCESS`; in case of failure, timer responds with `CANCEL_FAILURE` followed by `TIMEOUT`.

Finally, nondeterminism and state machines are also useful for modeling failures, consequences of physical processes occurring in the environment of a distributed system. Listing 2 shows how we model link failures using a model function `DoRPC`. Machine `Sender` uses this function to send a network message; the model nondeterministically either drops the message (and returns *false*) or sends the message (and returns either *true* or *false*).

Listing 3 shows how we model node failure using a model machine `FaultInjector`. This machine maintains a sequence `nodes` of machines for which we want to model *crash* failures; the initial sequence is passed to the machine when it is created and stashed in the variable `nodes`. The loop in state `Fail` chooses an index `i` in

Listing 4 Reliable, FIFO, no-dup channel.

```
event SEND: (dst:machine, val:any);
event RECV: (src:machine, val:any, seqNum:int);
event DELIVER: any;
machine Sender {
  var seqNumMap: [machine, int];
  var dst: machine;
  start state Init {
    on SEND do SendHandler;
  }
  fun SendHandler() {
    dst = payload.dst;
    monitor SendRecvSpec, M_SEND,
            (src=this, dst=dst, val=payload.val);
    if !(dst in seqNumMap)
      seqNumMap[dst] = 0;
    seqNumMap[dst] = seqNumMap[dst] + 1;
    while (true) {
      b = DoRPC(dst, payload.val, seqNumMap[dst]);
      if (b) break;
    }
  }
  ...
}
machine Receiver {
  var client: machine;
  var seqNumMap: [machine, int];
  var src: machine;
  start state Init {
    entry {
      client = payload as machine;
    }
    on RECV do ReceiveHandler;
  }
  fun ReceiveHandler() {
    src = payload.src;
    if (src in seqNumMap &&
        seqNumMap[src] <= payload.seqNum)
      return;  // ignore duplicates
    seqNumMap[src] = payload.seqNum;
    send client, DELIVER, payload.val;
    monitor SendRecvSpec, M_DELIVER,
            (src=src, dst=this, val=payload.val);
  }
}
```

the range `[0,sizeof(nodes))` nondeterministically, enqueues the `halt` event to `nodes[i]`, and removes the `i`-th entry from `nodes`. To model unexpected failure, `nodes[i]` should not handle the event `halt`; therefore, it will be terminated as soon as `halt` is dequeued.

4 A P program: Four-part harmony

We illustrate how the four parts of a typical P program—implementations, abstractions, specifications, and test drivers— come together. We use the example of a simple protocol for sending messages reliably with no duplication and in FIFO order atop a potentially unreliable network channel. Such a reliable network channel is a useful building block used by other protocols, such as Chain Replication, as we show in Section 6.

1. Implementation: The protocol implementation is shown in Listing 4. It uses two machine declarations, `Sender` and `Receiver`. Each client machine that wants to use the protocol creates one instance each of `Sender` and `Receiver` to act as a proxy for sending and receiving messages, respectively. We use *host*, *sender*, and *receiver* to refer to this troika of machines. The machine *receiver* has a reference to its *host* machine and the *host* machine has a reference to its *sender* machine. When $host_1$ wants to send a message to $host_2$, it sends the message to $receiver_2$ instead.

The protocol uses three event declarations, `SEND`, `RECV`, and `DELIVER`. Events can have payloads associated with them, and the *send* operation can attach any value to a event that is a subtype of the payload type in the event declaration. On the sending end, *host* uses the `SEND` event to tell *sender* the message it wants delivered. The `SEND` event has a named-tuple payload attached to it; the fields `dst` and `val` are the destination machine and value to be sent, respectively. The type `any` of `val` is an abstract type that can be instantiated with any concrete type at runtime.

The machine *sender* handles a `SEND` request in the function `SendHandler` which invokes `DoRPC` (Listing 2), a model of the RPC implementation provided by the OS. A loop in `SendHandler` repeatedly invokes `DoRPC` until `true` is returned. Clearly, this implementation can cause duplicate messages at the receiver end; therefore, increasing sequence numbers are used to guard against duplication. The send operation inside `DoRPC` attaches a named tuple comprising the sender id, the value being sent, and the sequence number as the payload to `RECV` event. The current sequence number for a particular destination machine is maintained in the map `seqNumMap`.

On the receiving end, *receiver* maintains the sequence number for a particular source machine in its own `seqNumMap`. It handles a `RECV` event by checking for duplicates, updating the sequence number in `seqNumMap`, and delivering the message to *host* using the `DELIVER` event.

2. Specification: Listing 5 shows a monitor that encodes the safety specification for the reliable network channel described above. The monitor `SendRecvSpec`, also written as a state machine, accepts two events `M_SEND` and `M_DELIVER`. The payload for both events is a named tuple comprising the source machine id, the destination machine id, and the value to be delivered. The monitor maintains in a map, `pending`, the sequence of values that have been sent but not received yet. The correctness checking enforced by this monitor is closely tied to how it is invoked in the implementation (Listing 4): First, the event `M_SEND` is sent to the monitor when *sender* receives a `SEND` request from its *host*; Second, the event `M_DELIVER` is sent to the monitor when *receiver* delivers a value to its *host*; Third, the assertion in the handler for `M_DELIVER` checks that the value being delivered matches the value at the first entry in the

Listing 5 Channel safety specification.

```
event M_SEND:
      (src: machine, dst: machine, val: any);
event M_DELIVER:
      (src: machine, dst: machine, val: any);
monitor SendRecvSpec {
  var pending:
      map[(src: machine, dst: machine), seq[any]];
  var key: (src: machine, dst: machine);
  start state Init {
    on M_SEND do {
      key = (src=payload.src, dst=payload.dst);
      pending[key] +=
        (sizeof(pending[key]), payload.val);
    }
    on M_DELIVER do {
      key = (src=payload.src, dst=payload.dst);
      assert pending[key][0] == payload.val;
      pending[key] -= (0, payload.val);
    }
  }
}
```

pending sequence.

In addition to safety, P can also check liveness specifications. A violation to a liveness property is an infinite execution of the program. Infinite executions may arise in a P program because of nondeterministic choices in abstractions, e.g., the loop in `SendHandler` (Listing 4) may not terminate if every invocation of `DoRPC` returns false. Not all infinite executions are erroneous though. An infinite execution that happens because of unfairness either in process scheduling or in resolving nondeterministic choice is simply a modeling artifact. The liveness checker in P performs fair scheduling of machines; furthermore, the programmer may specify fair nondeterministic choices using `$$` (replacing `$` with `$$` in the code of `DoRPC`). The use of `$$`, as opposed to `$`, indicates that if the choice is made infinitely often, it must resolve to both true and false infinitely often.

Fairness allows P programmers to specify the space of valid infinite executions compactly. Among these executions, the specification of erroneous infinite executions is done via a liveness monitor (Listing 6). The monitor `SendRecvSpecLiveness` captures the property that any send request made to the `Sender` machine is eventually delivered to the `Receiver`. An instance of the monitor is created with a particular sequence number as the payload. The monitor waits in the state `WaitForRecv` for an event indicating that the message corresponding to that sequence number has been delivered to `Receiver`. This state is annotated as `hot` to indicate that it is an error to remain in this state infinitely often.

With only hot states, a monitor can specify "eventually" properties, i.e., something good happens eventually. To generalize liveness monitors to "infinitely-often" properties and in fact to all ω-regular properties [30], we introduce the annotation `cold` on states as well. A monitor with both hot and cold states specifies an erroneous execution as one which visits some hot state infinitely

Listing 6 Channel liveness specification.

```
monitor SendRecvSpecLiveness {
  var seqNum: int;
  start state SendDone {
    entry {
      seqNum = payload as int;
      raise UNIT;
    }
    on UNIT goto WaitForRecv;
  }
  hot state WaitForRecv {
    on M_RECV do {
      if (seqNum == payload)
        raise UNIT;
    };
    on UNIT goto RecvDone;
  }
  state RecvDone {
  }
}
```

often and visits all cold states only finitely often.

3. Abstraction: In Listing 7, we show `SenderAbs` and `ReceiverAbs`, abstractions of `Sender` and `Receiver` respectively, from the point of view of the client of the protocol. These abstractions are considerably simpler than the implementations. The machine `SenderAbs` sends the message directly to the destination without using RPC or sequence numbers; the machine `ReceiverAbs` simply forwards the messages it receives to the host.

To scale systematic testing to complex protocol stacks, when testing a protocol that is a client of the network layer, we would like to use the abstractions `SenderAbs` and `ReceiverAbs` rather than the implementation machines `Sender` and `Receiver`. The P language provide a variation on the `new` primitive to achieve this substitution. The constructor "`new A(e) for B`" tells the P compiler to use the machine `A` for systematic testing but the machine `B` for real execution. This language feature allows a client of the network layer to be programmed using: "`new SenderAbs for Sender;`" and "`new ReceiverAbs(this) for Receiver;`". An imple-

Listing 7 Abstraction of a reliable channel.

```
model SenderAbs {
  start state Init {
    on SEND do {
      send payload.dst, RECV,
           (src=null, val=payload.val, seqNum=0);
    };
  }
}
model ReceiverAbs {
  var client: machine;
  start state Init {
    entry {
      client = payload as machine;
    }
    on RECV do {
      send client, DELIVER, payload.val;
    };
  }
}
```

Listing 8 Channel driver.

```
event Unit;
main model Test {
  var i: int;
  start state Init {
    entry {
      sender = new Sender();
      receiver = new Receiver(this);
      new SendRecvSpec();
      i = 0;
      raise Unit;
    }
    on Unit goto SendReq;
  }
  state SendReq {
    entry {
      if (i == 10) return;
      i = i + 1;
      send sender, SEND, (dst=receiver, val=$);
      raise Unit;
    }
    on Unit goto SendReq;
    on DELIVER { };
  }
}
```

mentation can have more than one behavioral abstractions and a client can choose a particular abstraction pertinent to its view of the implementation.

4. Test driver: To check the code of `Sender` and `Receiver` against its specification `SendRecvSpec`, we need to create a test driver that models a client of the protocol. Listing 8 shows such a test driver specified as a model machine `Test`. The `main` keyword attached to `Test` indicates that execution starts with a single instance of `Test`. This instance creates a sender and a receiver machine and sends 10 `SEND` requests to the sender. Thus, `Test` specifies a bounded-input nondeterministic program, whose executions depend on machine scheduling choices and explicit nondeterministic choices made by `$`. The testing tool accompanying P systematically explores all the executions created by these choices looking for violations to the assertion inside the monitor `SendRecvSpec`. If a violation is found, a fully reproducible error trace is provided to the programmer.

5 P toolchain

We describe the runtime, compiler,and tools that make it easy to deploy a P service on a cluster.[1] The compiler converts every state machine in the P program into a state machine represented in C. The runtime executes the P program according to the program's operational semantics by using the C representation of the state machines. Thus, the compiler and runtime together ensure that the runtime behavior of a P program matches the semantics validated by the systematic testing.

[1]Together, we call these P tools *Fulliautomatyx*, a tip of the hat to the epic works of Goscinny and Uderzo.

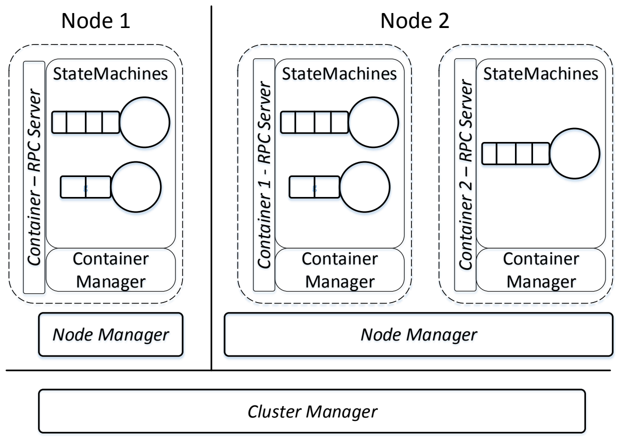

Figure 2: Components of P runtime

5.1 Runtime

Figure 2 shows the structure of a P service executing on a cluster of nodes. Each node hosts a collection of *Container* processes. Containers are collection of P state machines that interact closely with each other and reside in a common fault domain. Each node runs a *NodeManager* process which listens for requests to create new Container processes. Similarly, each Container hosts a single *ContainerManager* that manages the state machines in that Container. In the common case, each node has one NodeManager process and one Container process executing on it, but P also supports a collection of Containers per node enabling emulation of large-scale services running on only a handful of nodes.

A state machine can create another state machine in a new Container, on a remote node, by invoking `new` with appropriate parameters. The caller expects a new state machine *id* in return. This call is implemented by contacting the *ClusterManager*. The ClusterManager in turn contacts an appropriate NodeManager to create a new Container process and a new state machine within it. The *id* of a state machine running in a Container is a 3-tuple of the form <ip, port, mid>, where ip is the NodeManager's IP address, port is the Container's unique RPC server port number, mid is the state machine's unique ID withing the Container.

Each Container process hosts an RPC server which is used to forward P events to its state machines, including the ContainerManager. The ContainerManager handles `CreateMachine` events whose payload contains the state machine (*SM*) to be created and the value to be passed to the SM upon creation. It instantiates SM, together with its accompanying sender and receiver state machines (network protocol), and returns the id of the receiver to the source of the event.

The Container runtime, implemented in 7K lines of

C code, has the capability to (1) create, destroy and execute state machines, (2) create and destroy runtime representations of P types and values, (3) start an RPC server to listen to events from machines in a different Container, and (4) serialize data values before send and deserialize them after receive.

Although the operational semantics of a P program is that all machines run concurrently, the runtime does not use a separate thread for running each machine inside a Container. Rather, the machines are run in the context of a thread from the thread pool that services the Container's RPC server. Any remote procedure call to a Container results in a thread invoking a function to enqueue an event in the appropriate state machine running inside the Container. This function not only enqueues the event but also executes the event handling loop of the target state-machine. In this loop, events are dequeued and their handlers are executed to completion. If the handler enqueues an event into another machine in the same Container, the event handling loop of that machine is executed, and so on. If the handler tries to send an event to a machine in a different Container, the message is serialized and an RPC call is performed. The runtime uses efficient fine grain locking to guard against race conditions where more than one thread tries to execute a handler for a machine; such races, if allowed, would violate the operational semantics of P.

5.2 Compiler

The P compiler converts the source-level syntax of a P program into C code. This code contains separate statically-defined C array-of-structs for the events, machines, states, and transitions in the program. It also contains translations of all event handlers as C functions; the link between states and transtions and the appropriate handler is established via function pointers stored inside the C array-of-structs. The compiler is written using 3.5K lines of C# code and 3.3K lines of Formula [17] code.

The P compiler does not generate C code for monitors, model machines, and model functions. In cases where model functions and model machines are used for capture the interaction with the environment (e.g., *timer* and *RPC send* in Section 3), appropriate stub functions are generated for the programmer to provide a concrete implementation of these interactions.

The P compiler provides a simple module system that enables code reuse. A large program comprising many different protocols can be split into multiple P source files using the `include` directive. Starting from a root file, a preprocessor collects all dependent source files recursively, ensuring that each file is included exactly once.

5.3 Systematic testing

Our tool for systematic testing of P programs is implemented in two parts. First, in addition to generating C code for execution, the P compiler also generates a translation of the program into the Zing modeling language [4]. This part of the compiler comprises 3.5K lines of C# code for translating the P program and 1.5K lines of Zing code for modeling the semantics of the P runtime. Next, the Zinger tool takes as input the generated Zing model and systematically enumerates executions resulting from scheduling and explicit nondeterministic choices. Together, these two steps create a single-process interpreter and explorer for the nondeterministic semantics of a P program.

Clearly, the search problem faced by Zinger is enormous, even for a P program with a bounded-input driver (Listing 8). For example, if the driver creates n P machines and terminates in at most k steps, the number of possible executions is on the order of n^k. We use many techniques to combat this combinatorial explosion. The most important and a key contribution of this paper is compositional testing enabled by the abstractions specified by the programmer (e.g., Listing 7). There are other important optimizations [12] for searching executions of concurrent programs that are implemented in the Zinger tool itself; we exploit all of them to make efficient use of bounded testing resources.

As we have indicated before, P allows checking of liveness monitors as well. Liveness checking is considerably more difficult than safety checking. A safety violation is detected via reachability of a bad state, but a liveness violation is detected via reachability of a bad cycle that satisfies all fairness conditions [9]. To detect bad reachable cycles, Zinger implements iterative deepening over two algorithms—nested DFS [11] and maximum accepting predecessor [7]. We exploit this capability in Zinger to enable validation of general liveness monitors.

6 Building & testing distributed services compositionally

So far, we have seen the implementation, abstraction, specification, and driver of a single protocol for network transmission. To build a real distributed application, in general, many other protocols will need to be implemented, potentially stacked on top of the network layer.

In P, building these protocols compositionally, a practice advocated for networked systems [27, 33], and using abstractions in place of implementations in the protocol tree, enables developers to scale systematic testing. Furthermore, two protocols that share a common abstraction, and that are tested independently, can be used interchangeably by a client that is tested against their common abstraction.

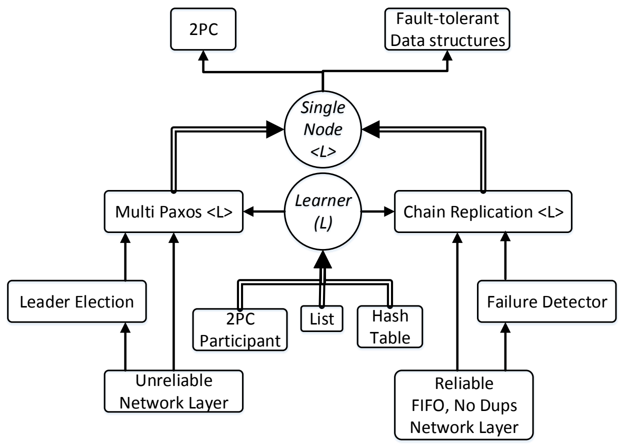

Figure 3: Protocol Composition. Circles are abstractions. $A \rightarrow B$ indicates that B uses A. $A \Rightarrow B$ indicates that A implements B.

Figure 3 shows a collection of protocols that we implemented in P with the goal of enabling two distributed services: (i) distributed atomic commit of updates to decentralized, partitioned data using two-phase commit [5, 13, 24, 26], and (ii) distributed data structures such as hashtables and lists. These distributed services use State Machine Replication (SMR) for fault-tolerance [14, 21, 29]. Protocols for SMR, like Multi-Paxos and Chain Replication, in turn use other protocols like leader election, failure detectors, and network channels.

Each protocol shown above has an abstraction that is used in testing its client. For example, in Section 4, we showed the abstraction for reliable channels, used here by other protocols. In the remainder of this section, we focus on `SingleNode` and `Learner` abstractions.

Single node abstraction: SMR protocols, such as Multi-Paxos [22] and Chain Replication [32], are used to make services fault-tolerant. For example, the Paxos Commit protocol for distributed transaction commit, proposed by Gray and Lamport, uses SMR to make two-phase commit fault-tolerant [14].

To understand why Muti-Paxos and Chain Replication use different sub-protocols (Figure 3), it is important to understand the settings they are designed for. Multi-Paxos is designed for asynchronous setting (not to be confused with non-blocking operations) and it uses $2n+1$ replicas to tolerate n failures. On the other hand, Chain Replication exploits a failure detector to use only $n+1$ replicas for tolerating n failures. Although Chain Replication is designed for a synchronous, fail-stop setting [28], where reliable detection of crash failures is possible, it has built-in safeguards that ensure correctness

Listing 9 Single node abstraction.

```
event REQ: (client: machine, reqid: int, op: any);
event RESP: (reqid: int, val: any, commitid: int);
model SingleNode<L> {
  var learner: machine;
  var commitid: int;
  var pending: seq[(client: machine, reqid: int)];
  start state Init {
    entry {
      if (learner == null) learner = new L();
      // respond to a prefix of requests
      while ($ && 0 < sizeof(pending)) {
        // respond to first request and remove
        send learner, REQ, (client=this,
                            reqid=pending[0].reqid,
                            op=pending[0].op);
        push WaitForResponse;
        pending -= 0;
        commitid = commitid + 1;
      }
    }
    on REQ goto Init with {
      // put request at nondeterministic position
      if ($) {
        pending += (Choose(),
                    (client=payload.client,
                     reqid=payload.reqid));
      }
    };
  }
  fun Choose(): int {
    var i: int;
    i = 0;
    while ($ && i < sizeof(pending)) {
      i = i + 1;
    }
    return i;
  }
  state WaitForResponse {
    on RESP do {
      send pending[0].client, RESP,
           (reqid=payload.reqid, val=payload.val,
            commitid=commitid);
      pop;
    };
  }
}
```

even with unreliable failure detection in an asynchronous real-world setting. With such safeguards, such as membership view-changes and dropping messages from older views, an imperfect failure detector can only affect performance and not correctness. The reliability of failure detectors can be enhanced by using *spies* across various layers of the systems stack to accurately detect failures and kill if needed [2, 25].

Despite their differences, both Multi-Paxos and Chain Replication ensure that the replicated state machine behaves logically identical to a single remote state machine that never crashes [16]. Listing 9 shows this abstraction as the model `SingleNode`, which maintains a sequence of pending requests. A fresh request is added at a nondeterministic position in the sequence but the responses are delivered in sequence order. To systematically test clients of SMR, such as two-phase commit and fault-tolerant data structures, we can ignore the specifics of SMR protocols and instead replace them with a `SingleNode` state ma-

Listing 10 Learner abstraction.

```
model Learner {
  var pending: seq[(client: machine, reqid: int)];
  start state Init {
    on REQ do {
      send payload.client, RESP,
           (id=payload.id, val=null, commitid=0);
    };
  }
}
```

chine. For example, a P two-phase commit implementation can use "`new SingleNode<Participant> for MultiPaxos<Participant>`"; this tells the P compiler to use `SingleNode` for systematic testing but `MultiPaxos` for real execution. Our compositional approach has the benefit of separating the validation of clients of SMR from the validation of Multi-Paxos and Chain Replication protocols. Thus, the problem of systematically testing clients, like two-phase commit, becomes much simpler.

The P implementations of Multi-Paxos and Chain Replication are validated separately from each other, each with its own protocol-specific safety and liveness specifications. In addition to those specifications, they share a common specification, adherence to which guarantees that they implement the `SingleNode` abstraction. Thus, clients tested with the `SingleNode` abstraction can interchangeably use Multi-Paxos and Chain Replication.

Learner abstraction: SMR protocols, such as Multi-Paxos and Chain Replication, ensure that the set of replicas agree on a common order of operations to be performed. The semantics of these operations is opaque to the replicas in these protocols. Each replica is associated with a learner state machine that executes these operations. Specific learners could be a hashtable that serves `PUT` and `GET` requests and maintains the hashtable state in memory or on disk. `Learner` in Listing 10 is an abstraction for this state machine that is independent of the semantics of the operations; it receives a `REQ` event and immediately responds to it with a `RESP` event. The systematic testing of both Multi-Paxos and Chain Replication uses this abstraction for the learner. The learner state-machine should be deterministic for the correctness of SMR protocols; thus for a sequence of `REQ`s, the learner should always produces the same sequence of `RESP`s and end up in the same state.

7 Evaluation

We evaluate P along three dimensions: 1) we compositionally test the protocols described in Figure 3 (Section 7.1), 2) we classify the bugs found by our framework (Section 7.2), and 3) we evaluate the performance of the distributed services built using P by deploying and benchmarking them on a cluster (Section 7.3).

Protocols Components	Impl.	Spec	Abstr-action	Driver	Total
2PC	230	90	88	30	438
Chain Repl.	360	220	90	130	800
Multi-Paxos	410	160	140	120	830
Basic Paxos	168	95	20	19	302
Leader Election	89	10	45	24	168
Dispatcher	138	65	48	37	288
Network Layer	75	25	30	21	151

Table 1: P LOC for different protocols.

Experimental setup: We perform all systematic testing on an Intel Xeon E5-2440, 2.40GHz, 12 cores (24 threads), 160GB machine running 64 bit Windows Server 2008 R2. Zinger's parallel iterative depth-first search algorithm efficiently uses multi-core machines. All experiments that benchmark distributed services built using P run on A2 VM instances on Windows Azure. The VMs have a 2-core Intel Xeon E5-2660 2.20GHz Processor, 3.5GB RAM, 2 virtual hard disks running 64 bit Windows Server 2012 R2.

Table 1 shows a four-part breakdown, in source lines of code, of our P implementations of all the protocols shown in Figure 3. *Dispatcher* here refers to front-end nodes that forward client requests to appropriate nodes in a replica group.

7.1 Compositional Testing

When testing a protocol in isolation, the P compiler composes it with the appropriate abstractions of other protocols and external libraries it interacts with, as specified by the programmer. Multiple test drivers for each protocol test for various corner cases and scenarios. The P framework automatically tests each component and produces a counter example if a bug is found. The counter example is a sequence of global interactions that lead to an error state.

Table 2 presents the number of bugs caught by our testing framework. We report only hard-to-find bugs that take Zinger more than a minute to find and ignore simple errors like invalid type-cast or variable initialization which are caught easily (within few seconds). Using the search prioritization techniques implemented in Zinger [12], we are able find most of these bugs within few minutes. This allows developers to go through many iterations of protocol development and testing, with each iteration converging on protocol correctness with respect to a specification and test driver. Without the compositional approach, Zinger fails to find these bugs in the allocated time budget of 2 hours.

After fixing all the bugs that could be quickly caught by the testing framework, we verify each protocol compositionally (for a given finite test driver). Table 3 shows the amount of time taken by Zinger to exhaus-

Protocols Components	Unhandled Event	Spec. Violation Safety	Spec. Violation Liveness
2 PC	20	15	4
Chain Repl.	26	10	7
Multi-Paxos	10	22	3
Basic Paxos	6	6	-
Leader Election	9	-	-
Dispatcher	10	8	1
Network Layer	2	1	-
Total	83	62	15

Total bugs found = 160

Table 2: Bugs found during the development process

Proto.	Safety Spec	Liveness Spec (for $\leq N$ failures)
2PC	*Atomicity*	Coordinator makes progress
Chain Repl.	All invariants in [32], cmd-log consistency	With stable head node, every cmd is eventually responded to
Multi-Paxos	All consensus requirements in [23], log consistency [31]	With stable leader, every proposal is always eventually learnt by learners
Basic Paxos	All consensus requirements in [23]	With distinguished proposer, a proposal is eventually chosen

Table 4: Example specifications checked for each protocol. SMR protocols are configured to be N fault-tolerant

tively search the entire state space for different protocols. Though we exhaustively explore protocols with only small number of nodes (3 to 7), it still gives enough confidence about the correctness of the system. Without using the compositional approach, the state space explodes faster and Zinger is unable to finish exploring the state space in 24 hours and runs out of memory.

7.2 Classification of bugs

The bugs we found during the development process of all the distributed protocols can be classified into the two categories:

1. Unhandled event violations: If an event e arrives in a state n, and there is no transition defined for e, then the verifier flags an *unhandled event* violation. There are certain circumstances under which the programmer may choose to delay handling of specific events or ignore the events by dropping them. These need to be specified explicitly so that they are not flagged by the verifier as unhandled. In our experience these kinds of bugs appear either when the programmer forgets to handle a particular event appropriately in a state or makes a false assumption about the possible events that other state machines can send. Bugs of these kind can lead to a *system crash* because of an unhandled event.

2. Specification violations: We implement both safety and liveness specifications of all the protocols as described in their respective papers [13, 22, 24, 32]. Table 4 shows examples of specifications checked for some of the distributed protocols. In our experience, writing these protocol level specifications along with the correct abstractions for compositional testing helps in better understanding of the system. We found severe correctness bugs in our implementations which were caught by the safety and liveness monitors (Table 2). Some of these bugs require subtle interleaving of the failure injector (multiple failures) which would have been hard to capture in a simulated testing environment.

Protocols (total number of nodes)	Time for exhaustive state space exploration (hh :mm)
2PC (5)	5:03
Chain Repl. (7)	4:02
Multi-Paxos (7)	10:39
Basic Paxos (7)	3:48
Leader Election (5)	0:43
Dispatcher (3)	0:20
Network Layer (3)	0:22

Table 3: Time for exhaustive exploration of distributed protocols.

7.3 Performance Evaluation

In this section, we evaluate the performance of the code generated by P for two distributed services: the fault-tolerant hashtable and fault-tolerant atomic commit. Specifically, we measure their peak throughput when using Multi-Paxos and Chain Replication.

Steady state throughput: To measure update throughput when there are no node failures in the system, clients pump in requests in a closed loop; on getting a response for an outstanding request, they go right back to sending another request. We scale the number of clients to ensure that services always have requests to process. Here we report numbers when using 3 clients.

The clients send their request to a dispatcher which maintains a versioned copy of replica membership. To make the services fault-tolerant, we use either Multi-Paxos or Chain Replication as described earlier. For Multi-Paxos, the dispatcher can send a request to any replica; replicas then forward requests to the distinguished leader. For Chain Replication, updates go the head of the chain. We configure Multi-Paxos and Chain Replication to use the HashTable and 2PC participant learners as appropriate.

Figure 4 shows the peak throughput, measured at the replica that responds to clients, for one and two fault-tolerant configurations. Thus, for Multi-Paxos, MP-1FT and MP-2FT have 3 and 5 replicas respectively. Similarly, for Chain Replication, CR-1FT and CR-2FT have 2 and 3 replicas respectively.

We highlight two observations, consistent with the nature of the protocols. First, MP configurations have slightly lower throughput than their corresponding CR configurations, consistent with the rounds of message exchanges in these protocols. Second, 2FT configurations are slower than their corresponding 1FT configurations

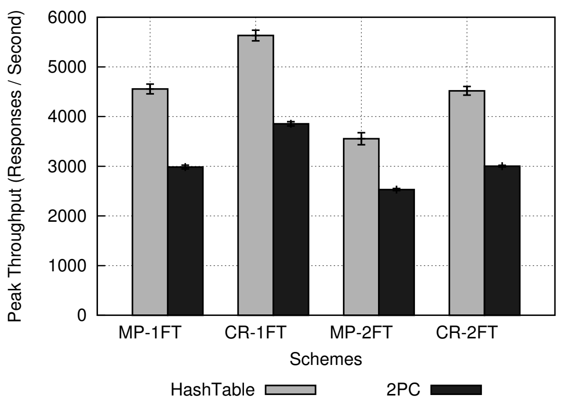

Figure 4: Peak fault-tolerant HashTable and 2PC throughput when using Multi-Paxos (MP) and Chain Replication (CR) configured to tolerate 1 and 2 faults (mean, SD over three 60s runs).

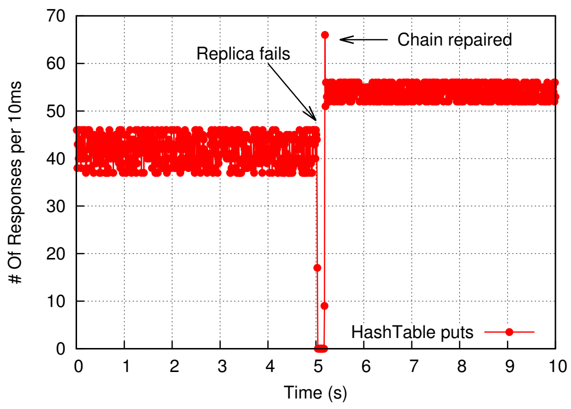

Figure 5: Timing diagram showing effect of a replica failure on HashTable put throughput when using a replica set with Chain Replication.

due to more state machines involved in the critical path of a command being processed.

Throughput during reconfiguration: Figure 5 shows the throughput (responses every 10ms), over a 10 second interval, of a fault-tolerant hashtable using Chain Replication with 3 replicas. At the 5 seconds mark, a replica failure causes a stall in update processing. Once the chain is repaired the throughput shoots back up, initially processing buffered requests, and quiesces at a higher rate than before as one lesser replica is in the critical path of update processing.

8 Related Work

We have discussed related work throughout the paper. In this section, we pay particular attention to those efforts that are most closely related to the techniques used in P. We begin with acknowledging SPIN [1] and many other model checkers as the inspiration for the search techniques underlying P, particularly those related to liveness checking.

The Mace [18, 19] and P [10] programming languages are closest in spirit to the goals of P. Similar to our work, both languages are based on the computational model of communicating state machines and provide a tool for exploring program executions. The most important advance made by our work is scalable compositional testing enabled by first-class support for abstractions as model machines. There are also many differences in the language primitives for writing specifications. Neither Mace nor P support explicit safety and liveness monitors. In particular, P allows the programmer to test for violations to any ω-regular liveness specification.

MODIST [35] was the first system to provide support for systematic testing of existing distributed systems programmed in C/C++. The Demeter [15] system improves the scalability of MODIST by exploiting the structure of a large distributed system as a collection of protocols. While exploring the executions of the entire system, Demeter attempts to infer interfaces for each component protocol. In contrast, P allows the programmer to specify these interfaces as model machines and the compiler automatically replaces the implementation with the abstraction during systematic testing. In our experience, specifying the interfaces for protocols in a large system has the important benefit of making explicit the assumptions protocols make about each other; in our approach, these assumptions live in the code as model machines and act as formal documentation. To get a better understanding of the tradeoffs, an interesting avenue for future work would be to compare the interfaces inferred automatically by Demeter with those specified by the programmer.

9 Conclusion

P is a new approach that makes it easier to build, specify, and test distributed systems. We used P to design and implement two distributed services that combine ten different protocols. Our experience indicates that the use of P significantly improved our productivity in arriving at a correct design.

10 Acknowledgements

This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] The model checker SPIN. *IEEE Trans. on Software Engineering*, 1997.

[2] M. K. Aguilera and M. Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, Berkeley, CA, USA, 2009. USENIX Association.

[3] B. Alpern and F. B. Schneider. Defining liveness. Technical report, 1984.

[4] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*. 2004.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[7] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In *FMCAD*. 2004.

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

[9] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *CAV*. 1991.

[10] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of PLDI*, 2013.

[11] A. Desai, S. Qadeer, S. Rajamani, and S. Seshia. Iterative cycle detection via delaying explorers. Technical Report MSR-TR-2015-28, Microsoft Research, 2015.

[12] A. Desai, S. Qadeer, and S. Seshia. Systematic testing of asynchronous reactive systems. Technical Report MSR-TR-2015-25, Microsoft Research, 2015.

[13] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978.

[14] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006.

[15] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 265–278, 2011.

[16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[17] E. K. Jackson. A module system for domain-specific languages. *TPLP*, 14(4-5):771–785, 2014.

[18] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 179–188, 2007.

[19] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Symposium on Networked Systems Design and Implementation*, 2007.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[22] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[23] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), Dec. 2001.

[24] B. W. Lampson and H. E. Sturgis. Crash recovery in a distributed data storage system. In *Technical Report, Computer Science Laboratory, Xerox, Palo Alto Research Center*, Palo Alto, CA, 1976.

[25] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 279–294, New York, NY, USA, 2011. ACM.

[26] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Trans. Comput. Syst.*, 1(1): 3–23, Feb. 1983.

[27] J. Saltzer, D. Reed, and D. Clark. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems*, 2:277–288, Nov. 1984.

[28] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1 (3):222–238, Aug. 1983.

[29] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[30] W. Thomas. Handbook of theoretical computer science (vol. b). chapter Automata on Infinite Objects, pages 133–191. MIT Press, Cambridge, MA, USA, 1990.

[31] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3), Feb. 2015.

[32] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[33] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Softw. Pract. Exper.*, 28:963–979, July 1998.

[34] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[35] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[36] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 54–66, London, UK, 1999. Springer-Verlag.