Composable Reliability for Asynchronous Systems

Sunghwan Yoo[1,2] Charles Killian[1] Terence Kelly[2] Hyoun Kyu Cho[2,3] Steven Plite[1]
[1]*Purdue University* [2]*HP Labs* [3]*University of Michigan*

Abstract

Distributed systems designs often employ replication to solve two different kinds of availability problems. First, to prevent the loss of data through the permanent destruction or disconnection of a distributed node, and second, to allow prompt retrieval of data when some distributed nodes respond slowly. For simplicity, many systems further handle crash-restart failures and timeouts by treating them as a permanent disconnection followed by the birth of a new node, relying on peer replication rather than persistent storage to preserve data. We posit that for applications deployed in modern managed infrastructures, delays are typically transient and failed processes and machines are likely to be restarted promptly, so it is often desirable to resume crashed processes from persistent checkpoints. In this paper we present MaceKen, a synthesis of complementary techniques including Ken, a lightweight and decentralized rollback-recovery protocol that transparently masks crash-restart failures by careful handling of messages and state checkpoints; and Mace, a programming toolkit supporting development of distributed applications and application-specific availability via replication. MaceKen requires near-zero additional developer effort—systems implemented in Mace can immediately benefit from the Ken protocol by virtue of following the Mace execution model. Moreover, this model allows multiple, independently developed application components to be seamlessly composed, preserving strong global reliability guarantees. Our implementation is available as open source software.

1 Introduction

Our work matches failure handling in distributed applications to deployment environments. In managed infrastructures, unlike the broader Internet, crash-restart failures are common relative to permanent-departure failures. Moreover, correlated failures are more likely: Application nodes are physically co-located, increasing their susceptibility to simultaneous environmental failures such as power outages; routine maintenance will furthermore restart machines either simultaneously or sequentially. Our toolkit masks crash-restart failures, preventing such brief or correlated failures from causing data loss or increased protocol overhead due to application-level failure handling.

Traditional wide-area distributed systems employ replication to solve two different kinds of availability problems. First, to prevent the loss of data through the permanent destruction or disconnection of a node, and second, to allow prompt data retrieval when some nodes respond slowly. Persistent storage can protect data from crash-restart failures, but it must be handled very carefully to avoid replica consistency problems or data corruption. For example, recovering a key-value store node requires checking data integrity and freshness and forwarding data to the new nodes responsible for it if the mapping has changed. Recovery can be quite tricky, particularly as little is known of the disk and network I/O in progress when the failure occurred. Recovery is further complicated if multiple independently developed distributed systems interact. Given that replication will be used anyway to ensure availability, and because correctly recovering persistent data after failures is difficult, many distributed systems choose to handle crash-restart failures and timeouts by treating them as a permanent disconnection followed by the birth of a new node, relying on peer replication, rather than persistent storage, to preserve data.

As new applications are increasingly deployed in managed environments, one appealing approach is to deploy wide-area distributed systems directly in these managed environments. However, without persistent storage, a simultaneous failure of all nodes (e.g., a power outage) would destroy all data. A more modest failure scenario in which machines are restarted sequentially for maintenance may be acceptable for a distributed system that does not employ persistent storage, but only if the system can process churn and update peer replicas quickly enough that all copies of any individual datum are not simultaneously destroyed. Wide-area distributed systems, such as P2P systems, are therefore often not well-suited to tolerate the types of failures more likely to occur in managed infrastructures, despite being designed to tolerate a high rate of *uncorrelated* failures. If crash-restart failures can be masked, however, such systems can ignore challenging correlated failures while still providing replication-based availability. Additionally, as node departures are infrequent, and performance less variable across managed nodes, in some cases fewer replicas will suffice to meet availability requirements.

At the other end of the spectrum, some applications that run in managed infrastructures do not require strong availability—distributed batch-scientific computing applications can seldom afford replication because they often operate at the limits of available system memory. If a failure occurs in these applications, they wish to lose as little time to re-computation as possible. In the worst case, a computation can be restarted from the input data. If we can mask crash-restart failures, we remove a large class of possible failures for distributed batch computing applications in managed clusters, as the cluster machines are unlikely to fail permanently during any given batch run.

In this paper, we describe the design and implementation of Ken, a protocol that transforms integrity-threatening crash-restart failures into performance problems, even across independently developed systems (Section 3). Ken uses a lightweight, decentralized, and uncoordinated approach to checkpoint process state and guarantee *global* correctness. Our benchmarks demonstrate that Ken is practical under modest assumptions and that non-volatile memory will improve its performance substantially (Section 5.1).

We further explain how Ken is a perfect match for a broad class of event-driven programming techniques, and we describe the near-transparent integration of Ken and the Mace [20] toolkit, yielding MaceKen (Section 4). Systems developed for Mace can be run using MaceKen with little or no additional effort. We evaluate MaceKen in both distributed batch-computing environments and a distributed hash table (Sections 5.2 and 5.3), demonstrating how Ken enables unmodified systems to tolerate otherwise debilitating failures. We also developed a novel technique to accurately emulate host failures using Linux containers [26]—basic process death, e.g., from killing the process, causes socket failures to be promptly reported to remote endpoints, which does not occur in power failures or kernel panics. We show how enabling Ken for a system developed for the Internet can prepare it for deployment in managed environments susceptible to correlated crash-restart failures. Existing application logic to route around slow nodes will continue to address unresponsive nodes, while safely remaining oblivious to quick process restarts.

Finally, we illustrate a broader, fundamental contribution of the Ken protocol: the effortless composition of independently developed systems and services, retaining the same reliability guarantees when the systems interact with each other without coordination, even during failure. In this example, a hypothetical scenario involving auctions and banking, failures would normally lead to loss of money or loss of trade. Ken avoids all such problems under heavy injected failures (Section 5.4).

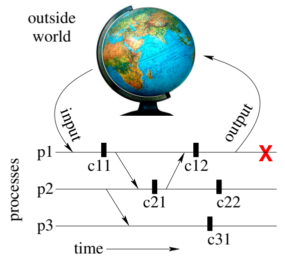

Figure 1: Abstract distributed computation

2 Background

Before describing the Ken protocol, we first review relevant concepts surrounding fault-tolerant distributed computing. Ken allows the developer to treat failed processes and hosts simply as slowly responding nodes, even across independently developed systems. We explain how Ken provides *distributed consistency*, *output validity*, and *composable reliability*.

To understand these concepts, consider Figure 1, illustrating standard concepts of distributed computing [12]. In the figure, time advances from left to right. Distributed computing processes p_1, p_2, and p_3 are represented by horizontal lines. Processes can exchange messages with each other, represented by diagonal arrows, and take checkpoints of their local state, represented by black rectangles. A crash, represented by a red "X," destroys process state, which may be restored from a checkpoint previously taken by the crashed process. Processes may also receive inputs from, and emit outputs to, the outside world. The outside world differs from the processes in two crucial ways: It cannot replay inputs, nor can it roll back its state. Therefore inputs are at risk of loss before being checkpointed and outputs are irrevocable.

2.1 Distributed Consistency

Checkpoints taken by two different processes are termed *inconsistent* if one records receiving a message the other does not record sending, because the message was sent *after* the sender's checkpoint was taken. Checkpoints c_{11} and c_{21} in the figure are inconsistent because c_{21} records the receipt of the message from p_1 to p_2 but c_{11} does not record having sent it. A set of checkpoints, one per process, is called a *recovery line* if no pair of checkpoints is inconsistent. A recovery line represents a sane state to which the system may safely be restored in response to failure. A major challenge in building durable systems lies in the efficient maintenance of recovery lines. If a process were simply to take checkpoints at fixed time intervals, some may not be suitable for any recovery line. A checkpoint is termed *useless* if it cannot legally be part of any recovery line. Checkpoint c_{21} is useless, as it is inconsistent with both checkpoints c_{11} and c_{12}.

One of the best-known approaches to constructing recovery lines is the Lamport-Chandy algorithm [5]. This algorithm requires distributed coordination, adding overheads, especially if checkpoints are taken frequently. Additionally, coordinated checkpoints may be impractical if independently developed/deployed applications are composed as discussed in Section 2.3.

2.2 Output Validity

Outputs emitted to the outside world raise special difficulties. Because the outside world by definition cannot roll back its state, we must assume that it "remembers" all outputs externalized to it. Therefore the latter may not be "forgotten" by the distributed system that emitted them, lest inconsistency arise. Distributed systems must obey the *output commit* rule: All externalized outputs must be recorded in a recovery line. In Figure 1, the output by process p_1 violates the output commit rule, and the subsequent crash causes the system to forget having emitted an irrevocable output.

Failures (crashes and message losses) may disturb a distributed computation. We say that a distributed system satisfies the property of *output validity* if the sequence of outputs that it emits to the outside world *could have been* generated by failure-free operation. Lowell et al. discuss closely related concepts in depth [25].

2.3 Composable Reliability

Even if individual applications support distributed consistency and output validity, these properties need not apply to the *union* of the applications when the latter interact. Composing together independently developed and independently deployed/managed applications is very common in practice. In such scenarios, the global guarantees of distributed consistency and output validity require maintaining a recovery line spanning multiple independently developed applications, coordinating rollback across independently managed systems to reach a globally-consistent recovery line, and globally enforcing the output commit rule across administrative domains. We show that Ken provides a *local* solution, maintaining recovery lines, enforcing output commit, and recovering from failures without cross-application coordination.

3 Reliability Mechanism

Below we describe the Ken protocol as we have implemented it, its programming model, and its properties. The name and the essence of the protocol are taken from Waterken, an earlier Java distributed platform that presents different programming abstractions [6, 17].

3.1 Protocol

Ken processes exchange discrete, bounded-length messages with one another and interact with the outside world by receiving inputs and emitting outputs. Incoming messages/inputs trigger computations with two kinds of consequences: outbound messages/outputs, and local state changes. Each Ken process contains a single input-handling loop, an iteration of which is called a *turn*.

Ken turns are *transactional*: either all of their consequences are fully realized, or else it is as though the message or input that triggered the turn never arrived. During a turn, outbound messages and outputs are buffered locally rather than being transmitted. At the end of a turn all such messages/outputs and local state changes caused by the turn are atomically committed to durable storage. On checkpoint success, the buffered messages/outputs become eligible for transmission; otherwise they are discarded and process state is rolled back to the start of the turn. The Ken protocol does not prescribe a storage medium; implementation-specific requirements of disaster tolerance, monetary cost, size, speed, density, power consumption, and other factors may guide the choice of storage. Ken simply requires the ability to recover intact all checkpointed data following any tolerated failure.

Messages from successful turns are re-transmitted until acknowledged. An acknowledgment indicates that the recipient has not only *received* the message but has also *processed it to completion*. The ACK assures the sender that the turn triggered by the message ended well, i.e., all of its consequences were fully realized and atomically committed. The sender may therefore cease re-transmitting ACK'd messages and delete them from durable storage. Message sequence numbers ensure FIFO delivery between each sender-receiver pair and ensure that each message is processed exactly once. Outside-world interactions may have weaker semantics than messages exchanged among the "inside world" of protocol-compliant Ken processes, because by definition the outside world cannot be relied upon to replay inputs or acknowledge outputs. Crashes may destroy an input upon arrival, and may destroy evidence of a successful output a moment after such evidence is created. Specific input and output devices and corresponding drivers that mediate outside-world interactions may be able to offer stronger guarantees than at-most-once input processing and at-least-once output externalization, depending on the details of the devices concerned [17]. Our Ken implementation allows drivers to communicate with a Ken process via `stdin` and `stdout`.

Recovery in Ken is straightforward. Crashes destroy the contents of local volatile memory. Recovery consists of restoring local state from the most recent checkpoint and resuming re-transmission of messages from successfully completed turns. Recovery is a purely local affair and does not involve any interaction with other Ken processes nor any message/input/event replay. Because Ken's transactional turns externalize their full consequences if and only if they complete successfully, a Ken process that crashes and recovers is indistinguishable from one that is merely slow.

Two sources of nondeterminism may affect Ken computations: *local* nondeterminism in the hardware and software beneath Ken's event loop, and nondeterminism in the *interleaving of messages* from several senders at a single receiver. Ken ignores both. A crash may therefore change output from what it would have been had the crash not occurred. Consider a turn that intends to output the local time but crashes before the turn completes. Following recovery, the time will be emitted, but it will differ compared with failure-free behavior. Next, consider a Ken process that intends to concatenate incoming messages from multiple sources and output a checksum of the concatenation. The order in which messages arrive at this process from different senders may differ in a crash/recovery scenario versus failure-free operation; as a result the checksum output will also differ. In both cases, crashes result in outputs that are *different* but not *unacceptable* compared with failure-free outputs. As there exists a hypothetical failure-free execution with the same outputs, output validity holds.

Two further examples illustrate how Ken's approach to nondeterminism is sometimes positively beneficial. First, consider an overflow-intolerant "accumulator" process that accepts signed integers as messages, and adds them to a counter, initially zero. If three messages containing `INT_MAX`, `1`, and `INT_MIN` arrive from different senders in that order, the `1` will crash the accumulator. Following recovery, the re-transmitted `1` may arrive *after* `INT_MIN`, averting overflow. Next, consider a Ken-based "square root server." Requests containing 4, 9, and 25 elicit replies of 2, 3, and 5 respectively. Unfortunately the server is unimaginative—e.g., it crashes when asked to compute $\sqrt{-1}$. Requests containing perfect squares, however, will continue to be served correctly whenever they reach the server between crashes caused by undigestible requests; mishandled requests impair performance but do not cause incorrect replies to acceptable requests. Wagner calls this guarantee *defensive consistency* [10]. Ken ensures defensive consistency provided that bad inputs crash turns *before* they complete (e.g., via assertion failures). Our simple examples represent the kinds of corner-case inputs and "Heisenbugs" that commonly cause problems in practice. Ken sometimes allows naturally occurring nondeterminism to work in our favor: forgiving recovery with zero programmer effort is a natural side effect of abandoning deterministic replay as a goal. See Lowell et al. for a detailed discussion of the potentials and limitations of approaches that leverage nondeterminism to "erase" failures [25].

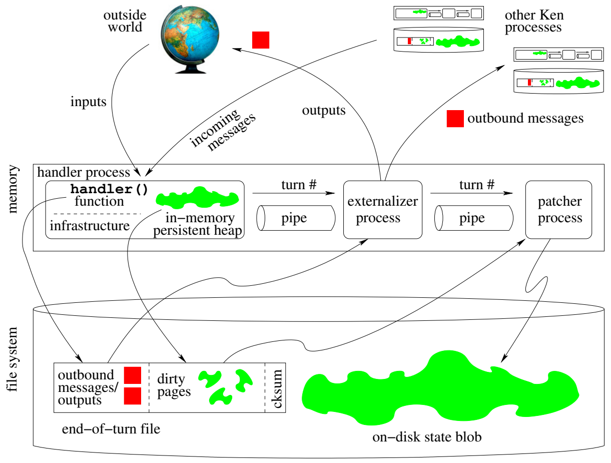

Figure 2: Ken internals

3.2 Implementation

Implementing generic support infrastructure for transactional event loops requires factoring out several difficult problems that would otherwise need to be solved by individual applications, e.g., efficient incremental checkpointing and reliable messaging. Furthermore it is not enough merely to provide these generic facilities separately; they must be carefully *integrated* to provide Ken's strong global correctness guarantees (distributed consistency and output validity). We describe first the programming model then the internal details of our Ken implementation in C for POSIX-compliant Unix systems such as Linux. Figure 2 illustrates the basic components and their flow of data.

Ken supports an event-driven programming paradigm familiar to many systems programmers and, thanks to JavaScript/AJAX, many more application programmers [29]. Whereas a conventional C program defines a `main()` function executed whenever the program is invoked, a Ken program defines a `handler()` function called by the Ken infrastructure in response to inputs, messages from other Ken processes, and alarm expirations. The handler may `send()` messages to other Ken processes, identified as network addresses, or emit outputs by specifying "`stdout`" as the destination in a `send()` call. The handler may also manipulate a persistent heap via `ken_alloc()` and `ken_free()` functions analogous to their conventional counterparts. The handler must eventually return (versus loop infinitely), and it may specify via its integer return value a time at which it should be invoked again if no messages or inputs arrive. A return value of -1 indicates there is no such timeout. An API allows application software to determine whether a given message has been acknowledged.

The Ken infrastructure contains the event loop that calls the application-level `handler()` function, passing inputs/messages as arguments. The sender of the message is also passed as an argument; in the case of inputs, the sender is "`stdin`." As the handler executes, the infrastructure appends outbound messages from `send()` calls to an end-of-turn (EOT) file whose filename contains the turn number. The infrastructure also manages the Ken persistent heap, tracking which memory pages have been modified: At the start of every turn the Ken heap is read-only. The first STORE to a memory page generates a segmentation fault; Ken catches the SIGSEGV, notes the page, and makes it writable. When the handler function returns, the infrastructure appends the turn's dirty pages to the EOT file along with appropriate metadata. Finally, Ken appends a 32-bit checksum to the EOT file.

As illustrated in Figure 2, a logical Ken process consists of three Unix processes: The *handler* process contains both the application-level handler function and most of the Ken infrastructure. The *externalizer* process re-transmits outbound messages until they are acknowledged. The *patcher* process merges dirty pages from EOT files into the *state blob* file, which contains the Ken persistent heap plus a few pages of metadata.

When the handler process concludes a turn, it sends the turn number to the externalizer via a pipe. The externalizer responds by `fsync()`ing both the EOT file and its parent directory, which commits the turn and allows the EOT file's messages/outputs to be externalized and its dirty pages to be patched into the state blob file; it also allows the incoming message that started the turn to be acknowledged. The externalizer writes outputs to `stdout` and transmits messages to their destinations in UDP datagrams. Messages to Ken processes are re-transmitted until acknowledged.

The externalizer tells the patcher a turn concluded successfully by writing the turn number to a second pipe. The patcher considers EOT files in turn order, pasting the dirtied pages into the state blob file at the appropriate offsets then `fsync()`ing the state blob. When all pages in an EOT file have been incorporated, and all messages in the EOT file have been acknowledged, the EOT file is deleted. As the patching process is idempotent, crashes during patching are tolerated and any state blob corruption caused by such crashes is repaired upon recovery.

Ken's three-Unix-process design complicates the implementation somewhat, but it carries several benefits. It decouples the handling of incoming messages, which generates EOT files, from the processing and deletion of EOT files. The `fsync()`s required to ensure durability occur in parallel with execution of the next turn, because the former are performed by the externalizer process and the latter in the handler process. If the handler process generates EOT files faster than the externalizer and patcher can consume them, the pipes containing completed turn numbers eventually fill, causing the handler process to block until the externalizer and patcher processes catch up.

Resurrecting a crashed Ken process begins by ensuring that all three of the Unix processes constituting its former incarnation are dead. A simple shell script suffices to detect a crash, thoroughly kill the failed Ken process, and restart it.[1] Ken's recovery code typically discovers that the most recent EOT file does not contain a valid checksum; the file is then deleted. The dirty pages in remaining EOT files are patched into the state blob file, which is then `mmap()`'d into the address space of the reincarnated handler process. We rely on `mmap()` to place the state blob at an exact address, otherwise pointers within the persistent heap would be meaningless. POSIX does not guarantee that `mmap()` will honor a placement address hint, but Linux and HP-UX kernel developers confirm that both OSes always honor the hint. In our experience `mmap()` always behaves as required. The externalizer process of a recovered Ken process simply resumes the business of re-transmitting unacknowledged messages.

3.3 Programming Guidelines

Ken programmers observe a handful of guidelines that follow from the abstract protocol, and our current implementation imposes a few additional restrictions.

The most important guideline is easy to follow: Write code as though tolerated failures cannot occur. Application programs running atop Ken never need to handle such failures, nor should they attempt to infer them. The most flagrant violation of output validity would be a Ken process that counts the number of times that it crashed and outputs this information. To provide a safe outlet for debugging diagnostics, we treat the standard error stream as "out of band" and exempt from Ken rules. Developers may use `stderr` in the customary fashion, with a few caveats: The three Unix processes of a logical Ken process share the same `stderr` descriptor, and to prevent badly interleaved error messages Ken applications should `write()` rather than `fprintf()` to `stderr`; furthermore `stderr` should pass through a pipe before being redirected to a file. Most importantly, `stderr` is "write-

[1] A Ken process that wishes to terminate *permanently* may convey to the resurrection script a "do not resuscitate" order via, e.g., an exit code, after confirming that sent messages have been acknowledged; terminating earlier would break the basic model and void Ken's warranties.

only." All bets are off if information written to stderr in violation of Ken's turn discipline finds its way back into the system.

Experienced programmers typically resist the next rule initially, then gradually grow to appreciate it: Deliberately crashing a Ken program is always acceptable and sometimes recommended, e.g., when corruption occurs and is detected during a turn. Crashing returns local Ken process state to the start of the turn, before the corruption occurred. Note that Ken substantially relaxes the traditional fail-stop recommendation that applications should try to crash *as soon as possible* after bugs bite [25]. Ken programmers may safely postpone corruption detection to immediately before the handler() function returns, i.e., Ken allows invariant verification and corruption detection to be safely consolidated at a single well-defined point. Assertions provide manual, application-specific invariant verification. Correia et al. describe a *generic* and *automatic* complementary mechanism for catching corruption due to hardware failures, e.g., bit flips [7].

Crashing a Ken program can do more than merely undo corruption. Memory exhaustion provides a good illustration of how crashing a Ken process can solve the root cause of a problem: The virtual memory footprint of a Unix process is the number of memory pages dirtied during execution, and a Ken process is no exception. Unlike an ordinary Unix process, however, Ken effectively migrates data in the persistent heap to the file system as a side effect of crash recovery. Upon recovery the Ken persistent heap is stored in the state blob file and the resurrected handler process contains only a read-only mapping, requiring no RAM or swap [30]. Persistent heap data will be copied into the process's address space on demand. Cold data consumes space in the *file system* rather than RAM or swap, which are typically far less abundant than file system space. A Ken program that calls assert(0) when ken_malloc() returns NULL thereby *solves the underlying resource scarcity problem*.

Ken applications must conform to the transactional turn model. Handler functions that cause externally visible side effects, e.g., by calling legacy library functions that transmit messages under the hood *during* a turn, void Ken's warranties. Conventional writes to a conventional file system from the handler function similarly break the transactional turn model because a crash between writes visibly leaves ordinary files in an inconsistent state. The preferred Ken way to store data durably, of course, is to use the persistent heap, though a basic filesystem driver could be implemented using Ken inputs and outputs.

Static, external, and global variables should be avoided because they are not preserved across crashes; Ken provides alternative means of finding entry points into the persistent heap. For example, Ken includes a hash table interface to heap entry points that is nearly as convenient as the static and global variables it is often used to replace. The biggest problem in practice is legacy libraries that employ static storage, e.g., old-fashioned non-reentrant random number generators and the standard strtok() function. In most cases safe alternatives are available, e.g., strtok_r(). The conventional memory allocator should not be used because the conventional heap doesn't survive crashes. Ken novices should limit themselves to the Ken persistent heap; knowledgeable programmers might consider, e.g., using alloca() for intra-turn scratch space.

Multithreading within a turn is possible in principle, but not recommended because Ken currently does not automatically preserve thread stacks across crashes. One easy pattern is guaranteed to work: Threads spawned by the handler function terminate before it returns. Trickier patterns involving threads that persist across handler invocations require more careful programming. Much of our own work explores shared-nothing message-passing computation, which plays to Ken's strengths, and we are often able to avoid the use of threads altogether.

Ken supports reliable unidirectional "fire and forget" messages, not blocking RPCs. We have not implemented RPCs for several reasons. First, they can be susceptible to distributed circular-wait deadlock whereas unidirectional messages are not. Furthermore output commit requires checkpointing all relevant process state prior to externalizing an RPC request, and in this case relevant state would include the stack, making checkpoints larger. More importantly, RPCs would disrupt Ken's "transactional turn" semantics as they externalize a request during a turn. In our experience it is often natural and easy to design a distributed computation in an event-driven style based on reliable unidirectional messages. The popularity of event-driven frameworks such as AJAX suggests that programming without RPCs is widely applicable.

A final area that requires care is system configuration. Most importantly, data integrity primitives such as fsync() must ensure durability. Storage devices often contain volatile write caches that do not tolerate power failures, which must be disabled. On some systems a small number of UDP datagrams can fill the default socket send/receive buffers; configuring larger ceilings via the sysctl utility allows Ken to increase per-socket buffers via setsockopt(), which reduces the likelihood of datagram loss. Other system parameters that sometimes reward thoughtful tuning are those that govern memory overcommitment and the maximum number of memory mappings. Multiple Ken processes running on a single machine should be run in separate directories for better performance.

3.4 Properties

Ken turns impose atomic, isolated, and durable changes on application state. If the application-level handler function always leaves the persistent heap in a consistent state when it returns—hopefully the programmer's intention!—then Ken provides ACID transactions that ensure local application state integrity. Ken also guarantees reliable pairwise-FIFO messages with exactly-once consumption. These benefits accrue without any overt act by the programmer; reliability is transparent.

By contrast, a common pattern in existing commercial software for achieving both application state and message reliability is to use a relational database to ensure local data integrity and message-queuing middleware to ensure reliable communications. In the RDBMS/MQ pattern it is the programmer's responsibility to orchestrate the delicate interplay between transactions evolving application data from one consistent state to the next and operations ensuring message reliability. The slightest error can easily violate global correctness, e.g., by overlooking the output commit rule or allowing a crash to introduce distributed inconsistencies. Transparent reliability is valuable even for relatively simpler batch scientific programs, where experience has shown that even experts find it very difficult to insert appropriate checkpoints [1].

When used as directed, Ken makes it *impossible* for the programmer to compromise distributed consistency or output validity. Distributed consistency in Ken follows directly from the fact that Ken performs an output commit atomically with every message sent and every output. The set of most recent per-process checkpoints in a system of Ken processes always constitutes a recovery line. Output validity follows from the fact that failures (message losses and/or process crashes) put a system of Ken processes into a state that could have resulted instead from message delays. More formal discussions of distributed consistency and output validity are available in [17].

Ken's most interesting property is *composable reliability*. Consider two systems of independently developed Ken processes. The two systems separately and individually enjoy the strong global correctness guarantees of distributed consistency and output validity. If they begin exchanging messages with one another, then the global correctness guarantees immediately expand to cover the *union* of the systems. The developers of the two systems took no measures whatsoever to make this happen. In particular they did not need to anticipate inter-operation between the two systems. Ken's reliability measures require no coordination among processes for checkpointing during failure-free operation, for recovery, or for output.

Ken furthermore brings important "social" benefits to software development. Because its reliability measures are purely local and independent, Ken *contains* damage rather than propagating it and *focuses* responsibility rather than diffusing it. For example, a crash of one Ken process does not trigger rollbacks of any other process; a remarkable number of prior rollback-recovery schemes do *not* have this property. The net effect is that Ken is unlikely to cause finger-pointing among teams responsible for designing and operating different components.

Finally, Ken is implementation-friendly in several ways. It is frugal with durable storage. Because recovery requires only the most recent local checkpoint, older checkpoints may be deleted. Ken never takes useless checkpoints [17]. Checkpoints are small as they include only the persistent heap, not the stack or kernel state; whole-process checkpoints taken at the OS or virtual machine monitor layer would be larger. An implementation may take checkpoints incrementally, as ours does. It is furthermore possible to delta-encode and/or compress checkpoints, though our current implementation does neither. Finally, Ken admits implementation as a lightweight, compact, portable library. Our stand-alone Ken implementation is available as open source software [16].

4 Event-Driven State Machine Integration

In this section, we describe integration of the Ken reliability protocol with an event-driven state machine toolkit. The concepts of common event-driven programming paradigms and Ken are complementary, allowing a seamless integration nearly transparent to developers.

4.1 Design

Event-driven programming has long been used to develop distributed systems because it matches the asynchronous environment of a networked system. In event-driven programming, a distributed system is a collection of event handlers reacting to incoming events. For example, events may be network events like message delivery, or timer events like a peer response timeout. To prevent inconsistency and avoid deadlock, event-driven systems frequently execute atomically, allowing a single event handler at a time. Event handlers are non-blocking, so programmers use asynchronous I/O, continuing execution as needed through dispatching subsequent events.

All execution therefore takes place during event handlers, and importantly, all outputs are generated therein. Typically, a single input is fed to the event handler, and it must run to completion without further inputs. To conform to the event loop, most event-driven toolkits contain specific I/O libraries for messaging—one that provides

```
while (running) {
  readyEvents = waitForEvents(sockets, timers);
  for (Event e in readyEvents) {
    if (Ken.isDupEvent(e)) { continue; }
    Ken.blockOutput();
    dispatchEvent(e); //becomes ken_handler()
    Ken.writeEOTFile();
    Ken.transmitAckAndOutputs(e);
} }
```

Figure 3: Common event loop with Ken integration

message I/O results only in subsequent events (i.e. fire-and-forget messaging).

Figure 3 shows a typical event loop for a distributed system. A single thread waits for network and timer events, then dispatches all ready events by calling their event handlers in turn. To integrate Ken, we need only verify that the input is new, block outputs by buffering them in the event library, and then acknowledge the input and externalize the outputs once the end-of-turn file is written to non-volatile storage. As there is only one thread, the EOT file will be consistent with the turn state. Finally, we replace the dispatch function with the Ken handler function, to provide access into the persistent heap.

4.2 Implementation

We now describe the integration of the Ken protocol with Mace [20], an open-source, publicly available distributed systems toolkit. To fully integrate Ken into Mace, we replaced the networking libraries with Ken persistent message handling and acknowledgments, replaced the facility for scheduling application timers with a Ken callback mechanism, replaced memory allocation in Mace with `ken_alloc()`, and connected the Ken `handler()` function to the Mace event processing code. Finally, we relinquished control over application startup to Ken.

Mace and Ken appeared to be a perfect fit for each other, as Mace provided non-blocking atomic event handlers, explicit persistent state definition, and fire-and-forget messaging. However, in the implementation integrating Mace and Ken we ran across numerous complicating details. Thankfully, these are largely transparent to the *users* of MaceKen, and need only be implemented in the MaceKen runtime. We now discuss a few of these.

State checkpoints. Mace provides explicit state definition, so we intended to checkpoint the explicit state only. However, many of the variables were collections based on the C++ Standard Template Library (STL), which internally handles memory management. This complicates checkpointing as the STL collections contained references to many dynamic objects. Instead, we replaced the global allocator, requiring all Mace heap variables to be maintained by Ken, even transient and temporary state. This exercise also caused us to streamline some runtime libraries to reduce the number of pages unnecessarily dirtied.

Initialization. Unlike Mace, Ken requires that the implementation of `main()` be defined within Ken, and not by a user program. This gives Ken control over application initialization, to set up Ken state appropriately without application interference, and also hiding application restart. As Mace allowed substantial developer flexibility on application initialization, this created some tension. We had to incorporate a MaceKen-specific initialization function that MaceKen would call on each start, to properly initialize certain state; however, this is hidden from users to preserve the MaceKen illusion that a program never fails. Ultimately, it makes both Mace and MaceKen easier to use—developers need not worry about complex system initialization.

Event Handlers. Mace provides atomic event handling by using a mutex to prevent multiple events from executing simultaneously. This design allows multiple threads to attempt event delivery, such as one set of threads delivering network messages, and another set of threads delivering timer expirations. This design is at odds with other common event dispatch designs where all event processing is done through a common event loop executed by a single thread. Ken assumes such a common, monolithic event loop, which required adding an event-type (e.g. message delivery, timer expiration, etc.) dispatch layer prior to the event handler dispatch Mace already used, adding additional overhead.

Transport Variants. By default Ken re-transmits messages until acknowledged, doubling the timeout interval with each re-transmission (i.e., exponential backoff). This strategy is based on the principle that in our target environments, network losses are infrequent and most retransmissions will be due to restarting Ken processes. However for communication-intensive applications, e.g., our graph analysis (Section 5.2), the volume and rate of communication increase the chances of loss due to limited buffer space in the network or hosts. To prevent excess retransmissions, we implemented two additional transport variants in addition to the original Ken default: First, we added Go-Back-N flow control atop the UDP-based protocol to minimize latency for message-intensive applications in situations where receive buffers are likely to fill. We have also implemented a TCP-based transport that simply re-transmits in response to broken sockets. The distributed graph analysis experiment of Section 5.2 and the distributed storage tests of Section 5.3 employ the TCP transport; the microbenchmarks of Section 5.1 used the default Ken mechanism.

Logging. Mace contains a sophisticated logging library that is not suitable for unbuffered `stderr` output.

As a result, we had to rewrite the library to specially use standard heap objects, in many cases replacing provided containers whose allocation we could not control. As with `stderr`, logging must be used as a write-only mechanism or MaceKen warranties are voided.

Our MaceKen implementation will be released as open source software [18].

5 Evaluation

We tested both our stand-alone Ken implementation and also MaceKen to verify that they deliver Ken's strong fault tolerance guarantees, to measure performance, and to evaluate usability.

5.1 Microbenchmarks

We conducted microbenchmark tests to measure Ken performance (turn latency and throughput) on current hardware and to estimate performance on emerging non-volatile memory (NVRAM). One or more pairs of Ken processes on the same machine pass a zero-initialized counter back and forth, incrementing it by one in each turn, until it reaches a threshold. The rationale for using two Ken processes rather than one is that our test scenario involves two reliability guarantees, local state reliability and reliable pairwise-FIFO messaging, whereas incrementing a counter once per turn in a single Ken process would not involve any of Ken's message layer. We ran our tests on a 16-core server-class machine with 2.4 GHz Xeon processors, 32 GB RAM, and a mirrored RAID storage system containing two 72 GB 15K RPM enterprise-class disks; the RAID controller contained 256 MB of battery-backed write cache. The storage system is configured to deliver enterprise-grade data durability, i.e., all-important foundations such as `fsync()` and `fdatasync()` work as advertised (our tests employ the latter, which is sufficient for Ken's guarantees).

We tested Ken in three configurations: the default mode in which `fdatasync()` calls guarantee checkpoint durability at the end of every turn; "no-sync" mode, in which we simply disable end-of-turn synchronization; and "`tmpfs`" mode, in which Ken commits checkpoints to a file system backed by ordinary volatile main memory rather than our disk-backed RAID array. The no-sync case allows us to measure performance for weakened fault tolerance—protection against process crashes only and not, e.g., power interruptions or kernel panics. The `tmpfs` tests shed light on what performance might be on future NVRAM.

We measure light-load latency by running only two Ken processes that pass a counter back and forth, incrementing it to a final value of 15,000 (150,000 for the `tmpfs` scenario). Default reliable Ken with `fdatasync()` averages 4.27 milliseconds per turn. Recall from Section 3.2 that Ken synchronizes twice at the end of each turn, once for the end-of-turn (checkpoint) file and once for the parent directory. The expected time for each call should roughly equal a half-rotation latency, which on our 15K RPM disks is 2 ms. Without end-of-turn synchronization, Ken's turns average 0.575 ms, roughly 7.4$\times$ faster; `tmpfs` further reduces turn latency to 0.468 ms.

The throughput of a single pair of Ken processes in our "counter ping-pong" test is limited by turn latency because the two Ken processes' turns must strictly alternate. To measure the aggregate turn throughput capacity of our *machine*, we vary the number of Ken processes running. As in our latency test, pairs of Ken processes increment a counter as it passes back and forth between them. With data synchronization on our RAID array, throughput increases gradually to a plateau, eventually peaking at over 1,750 turns per second when several hundred Ken processes are running. Without end-of-turn data synchronization, throughput peaks at over 6,000 turns per second when roughly sixteen Ken processes are running. On `tmpfs`, peak throughput exceeds 20,700 turns per second with 22 Ken processes running.

As expected, Ken's performance depends on the underlying storage technology and its configuration. Our enterprise-grade RAID array provides reasonable performance for a disk-based system. Our no-sync measurements show that latency drops substantially and throughput increases more than 3$\times$ if we relax our fault tolerance requirements. NVRAM would provide the best of both worlds: even lower latency and an additional 3$\times$ throughput increase over the no-sync case, without compromising fault tolerance. We have not yet conducted tests on flash-based storage devices but we expect that SSDs will offer substantially better latency and throughput compared to disk-based storage. At the other end of the spectrum, on a system with simple conventional disk storage, we have measured Ken turn latencies as slow as 26.8 ms.

For applications that must preserve data integrity in the face of failures, the important question is whether general-purpose integrity mechanisms such as Ken make efficient use of whatever physical storage media lie beneath them. In tests not reported in detail here, we found that Ken's transactional turns are roughly as fast as ACID transactions on two popular relational databases, MySQL and Berkeley DB. On a machine similar to the one used in our experiments, ACID transactions take a few milliseconds for both Ken as well as the RDBMSes. This isn't surprising because the underlying data synchronization primitives provided by general-purpose operating systems are the same in these three systems, and the underlying primitives dominate light-load latencies. Our finding merely suggests that gratuitous inefficiencies are absent from all three. Ken offers different features and ergonomic tradeoffs compared to relational databases—it provides reliable communications and strong global distributed correctness guarantees, but not relational algebra or schema enforcement, for example—and comprehensive fair comparisons are beyond the scope of the present paper.

5.2 Transparent Checkpoints: Distributed Graph Analysis

Recent work has applied Mace to scientific computing problems far removed from the systems for which Mace was originally intended [36]. In a similar vein, we further tested MaceKen's versatility by employing it for a graph analysis problem used as a high-performance computing benchmark [14]: The maximal independent set (MIS) problem. Given an undirected graph, we must find a subset of vertices that is both independent (no two vertices joined by an edge) and maximal (no vertex may be added without violating independence). A graph may have several MISes of varying size; the problem is simply to find any one of them.

We implemented a distributed MIS algorithm [31] that lends itself to MaceKen's event-driven style of programming. Like many distributed MIS solvers, this algorithm has a high ratio of communication to computation and so distribution carries a substantial inherent performance penalty. Our fault-tolerant distributed solver is actually *slower* than a lean non-fault-tolerant single-machine MIS solver when applied to random Erdős-Rényi graphs that fit into main memory on a single machine. However distribution is the only way to tackle graphs too large for a single machine's memory, and our MaceKen solver can exploit an entire cluster's memory. More importantly, our MaceKen solver can survive crashes, which is important for long-running scientific jobs.

To test MaceKen's resilience in the face of highly correlated failures, we ran our MIS solver on a graph with 8.3 million vertices and 1 billion edges on 20 machines and then simultaneously killed all MaceKen processes during the job. When we re-started the processes, the distributed computation resumed where it left off and completed successfully. We carefully verified that its output was identical to that of a known-correct reference implementation of the same algorithm. Our experiences strengthen our belief that MaceKen can be appropriate for scientific computing, and furthermore demonstrate that Ken can transparently add fault tolerance with zero programmer effort.

The largest single graph that our MaceKen MIS solver has tackled has 67 million vertices and 137 billion edges; a straightforward representation of this graph requires over 1.1 TB. Running on a 200-machine cluster, our MaceKen MIS solver took 8.96 hours to generate this random graph and 17.07 hours to compute an MIS with end-of-turn data synchronization disabled. Figure 4 compares this run time with the run times of a lean and efficient single-machine MIS solver on smaller graphs; all graphs are Erdős-Rényi graphs and the number of edges is 2048× the number of vertices. The single-machine solver is not based on Ken or MaceKen and it is not fault tolerant. Our distributed MaceKen solver can tackle graphs 8× larger than our single-machine solver. Figure 4 suggests that, given enough memory, the single-machine solver would probably run faster, but we do not have access to a single machine with 1.1 TB of RAM. Our results suggest that a MaceKen graph analysis running on sufficiently fast durable storage (NVRAM) can provide both fault tolerance and reasonable performance.

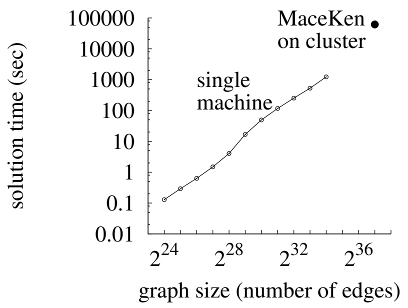

Figure 4: MIS: single machine vs. MaceKen cluster

5.3 Survivability: Distributed Storage

We conducted experiments on a Mace implementation of the Bamboo Distributed Hash Table (DHT) protocol [32] in the face of churn. No modifications were made to Bamboo to enable the Ken protocol—the Mace implementation compiled and ran directly with MaceKen. We chose to work with the Bamboo protocol because it was specifically designed to tolerate and survive network churn (short peer node lifespan). Bamboo uses a rapid join protocol and periodic recovery protocols to correct routing errors and optimize routing distance. Our own work [20, 19] has confirmed the results of others that Bamboo delivers consistent routing under aggressive churn. However, this work has focused on consistent *routing*, not the preservation of DHT data maintained atop Bamboo routing, which was expected to pose additional challenges and not be durable. The DHT is a separate implementation that uses Bamboo for routing, but is responsible for storage, replication, and recovery from failure. As the consistency and durability of data stored at failed DHT nodes on peer computers are suspect at best, traditional designs use in-memory storage only, and tolerate failures through replication instead. If a node reboots, it will rejoin the DHT and reacquire its data from peer replicas. Mace includes such a basic DHT, which we used for testing.

In exploring the Bamboo protocol's resilience to churn, we initially discovered that even under periods of relatively high churn (mean lifetime of 20 seconds), the Bamboo DHT is able to recover quickly and maintain the copies of stored data. While pleasantly surprised, we determined that this occurred as a direct result of the fast failure notification that surviving peers receive when a remote DHT process terminates and sockets are cleaned up by operating systems. However, in the case of power interruptions, kernel panics, or hardware resets, the socket state is *not* cleaned up but rather is erased with no notice at all. TCP further will not time-out the connection for several minutes after the surviving endpoint attempts to send data, delaying failure recovery. Once the physical machine has resumed operation, the OS will respond to old-socket-packets with a socket reset, causing the failure to be detected sooner. However, in both cases, failure is not detected unless the surviving endpoint attempts to send new data. As obtaining access to a large cluster where we can control the power cycling of machines is impractical, we sought to devise an alternate mechanism to conduct data survivability and durability tests.

Linux Containers (LXCs) [26] are a lightweight virtualization approach based on the concepts of BSD jails. Importantly, network interfaces can be bound within an LXC, with their own network stack of which the host operating system is unaware. We configured our experiment to use LXCs for running DHT nodes. For each LXC, a virtual Ethernet device is created, with one endpoint inside the LXC and the other in the host OS. The host OS then routes packets from the LXC to the physical network over the real Ethernet device. When we wished to fail a DHT node, we could first remove the host's virtual Ethernet device endpoint to prevent the network stack in the LXC from sending any packets. While killing the processes next caused attempts to cleanup the socket state, these failed due to lack of connectivity. Finally, destroying the LXC destroyed all evidence of the socket, allowing the LXC to be restarted without having TCP attempt to resend the socket FIN.

We conducted experiments to mimic failure scenarios likely to be observed in managed infrastructures. We ran 300 DHT nodes on 12 physical machines, using ModelNet [34] to emulate a low-latency topology with three network devices, and 100 DHT nodes connected to each. In our experiments, after an initial stabilization period, DHT clients would periodically put new data in the DHT, and request data, split between just-added data (Get) and previously-added data (Prior). Get requests commence ten minutes after start. Prior requests commence after 45 min to ensure that the DHT contains sufficient data. Our experimental setup places many DHT nodes on each physical machine, and if DHT nodes called `fsync()` the machine's storage system would be overloaded. We therefore emulate the latency of `fsync()` calls by adding 26 ms sleep delays. The slowest Ken *per-turn* latencies that we have observed are roughly 26 ms; since a Ken turn involves *two* `fsync()` calls, by adding 26 ms to each `fsync()` call our experiments measure Ken performance pessimistically/conservatively.

Since a DHT uses replication to increase availability and data survivability upon crashes, we have configured the unmodified Mace DHT implementation to have five replicas including the primary store. With Ken enabled, no replication is needed to survive crash-restart failures, so the MaceKen DHT stores data only on the primary store in these experiments.

In the middle of the experiment we tested two kinds of failures: first a "power interruption" that restarted all DHT nodes except a distinguished bootstrap node, and second a "rolling restart" that restarted each node twice over a period of 5 minutes, such as for urgent, unplanned maintenance to the entire cluster. Each restarted node is offline for only 5 seconds before being restarted—chosen to maximize the unmodified (i.e., non-Ken) Mace DHT implementation's ability to recover quickly—real operating systems currently take considerably longer to restart.

Figures 5 and 6 present success fractions of types of DHT lookups. For both experiments, when using the MaceKen runtime, no impact can be seen in the correct operation of the DHT storage, either for Get or Prior. When the unmodified Mace runtime is used, failures cause a period of disruption to DHT requests. When nodes failed simultaneously, after the period of disruption the Get requests resume delivering fast, successful responses, but most Prior requests fail due to permanent data loss when all replicas of the data simultaneously failed. In the rolling-restart case, some data survived because the DHT could detect failure of some replicas while other replicas still survived and could further replicate the data. If all machines failed before any of them detected failures and could replicate, then the data were lost.

Figures 7 and 8 show average latency for all the requests in each minute. MaceKen overheads roughly double the cost of the DHT lookups compared to Mace, but this is reasonable performance, particularly given the success fractions, and the slow storage device being simulated. During and immediately after the failures, MaceKen performance is slower because it is performing recovery, patching, and reliable data retransmission.

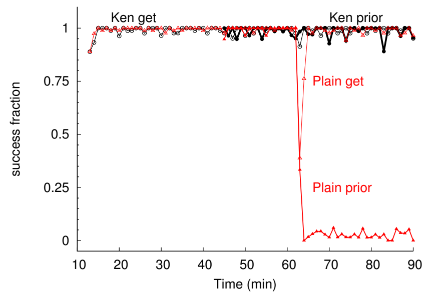

Figure 5: Simultaneous Failure

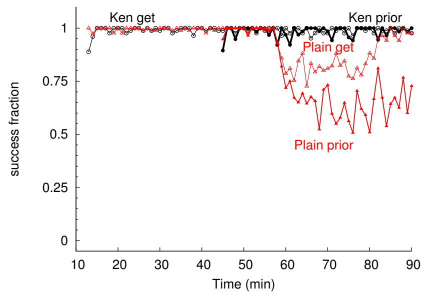

Figure 6: Rolling-restart

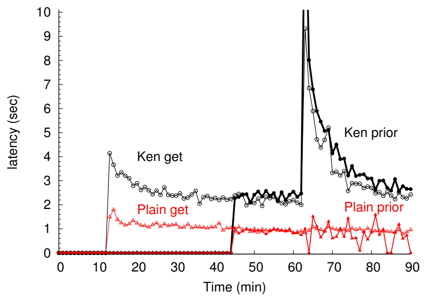

Figure 7: Simultaneous Failure

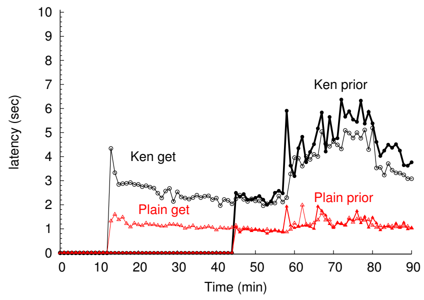

Figure 8: Rolling-restart

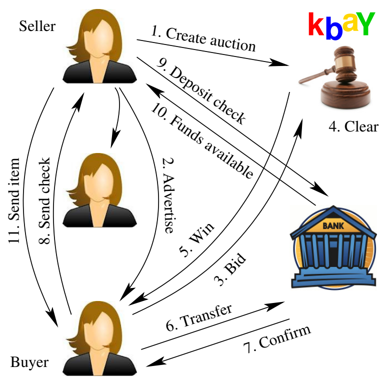

Figure 9: "kBay" e-commerce

As we made no modifications to Bamboo to use MaceKen, and based on Bamboo survivability in these experiments, we conclude that the MaceKen runtime is both easy to use and increases the survivability of existing peer-to-peer systems.

5.4 Composable Reliability: E-Commerce

Decentralized software development is the rule rather than the exception for complex distributed computing systems. To take a familiar example, "mashups" compose independently developed and managed Internet services in client Web browsers [35]. Other important examples, e.g., supply-chain management software, lack a single point of composition but nonetheless require end-to-end reliability across software components designed and deployed by teams separated by time, geography, and organizational boundaries. Ken is well suited to such systems because it guarantees reliability that composes globally despite being implemented locally.

Our "kBay" e-commerce scenario (Figure 9) stress-tests Ken's composable reliability. Sellers create auctions and advertise items for sale among friends, who bid on items. The kBay server clears auctions and notifies winners. Winning bidders must transfer money from savings to checking accounts before sending checks to sellers. Sellers deposit checks, causing a transfer from the buyer's checking account to the seller's savings account. If the check does not "bounce" the seller sends the purchased item to the buyer. Without Ken, crashes and message losses could create several kinds of mischief for kBay, e.g., causing the bank to destroy money or create counterfeit, causing the auction site to prematurely remove unsold items from the marketplace or award the same item to multiple buyers, causing checks to bounce or causing check writers to forget having written them. Similar problems have long plagued real banks [2] and e-commerce sites [3, 33].

Given complete control over all kBay software, a single careful development team could in principle guarantee global distributed consistency and output validity. Ken's composable reliability makes it easy for *separate* teams to implement components independently and still achieve global reliability without coordination. We implemented atop Ken all of the components depicted in Figure 9. In our tests, 32 clients offer items for sale via the auction server and advertise them among five other clients. Injected failures repeatedly crash the auction server, the bank, and the clients, which ran on three separate machines. We verified output validity by checking that every item is eventually sold to exactly one buyer and that the sum of money in the system is conserved. Our results confirm our expectations: Ken guarantees global correctness even in the presence of repeated crash/restart failures of stateful components designed without coordinated reliability.

6 Related Work

Previous work related to our efforts falls under three main umbrellas. First, there is currently a popular set of systems for managing cluster computation. These special case systems, while effective for their goals, are not general enough to support the range of applications we are targeting. Second, a host of toolkits for building varieties of distributed systems exist. However, these have typically targeted developing wide-area peer-to-peer systems. They do not provide the proposed combination of data center optimization, performance tuning, and reliability. Finally, the Ken design follows on a line of rollback-recovery research, applied to general purpose systems. Our proposed work shows how to apply the advances Ken makes in this line in a generic way to the development of a broad class of applications.

Cluster Computation Systems Cluster computing infrastructures include two broad classes. First, job scheduling engines such as Condor [22, 11] are designed to support batch processing for distributed clusters of computers. These schedulers tend to be focused on efficient scheduling of a large set of small-scale tasks, such as for a single machine, across a wide set of resources. More recently, systems such as MapReduce [8], Ciel [28] and Dryad [15], have emerged, and focus on how to partition single, large-scale data-parallel computations across a cluster of machines. Both classes support process failures, but the implementation is predominantly focused on batch processing. In batch processing, failure handling is much simpler, because only the eventual result is emitted as final output. Failures can therefore be tolerated by simply re-computing the result, possibly using cached partial earlier results.

MaceKen is suitable for developing non-batch applications that emit results continuously and for applications with continuous inputs and outputs. In non-batch applications, simply restarting crashed processes does not guarantee distributed correctness. In particular, we focus our design on approaches yielding distributed consistency and output validity, where the output remains acceptable despite tolerated failures. Consider, for example, the CeNSE application [23]. CeNSE includes a large group of sensors and actuators embedded in the environment, connected with an array of networks. These actuators provide near-real-time outputs, so the application's job is not to perform batch computations, but rather to generate outputs continuously. In contexts like CeNSE, output validity is critical.

Programming Distributed Systems There are many toolkits for building distributed systems. We built our system on top of the Mace toolkit, which includes a language, runtime, model checker, simulator, and various other tools [20]. But Mace, like many other toolkits, is focused on the class of general, wide-area distributed systems such as peer-to-peer overlays.

Other similar toolkits include P2 [24], Libasync and Flux [27, 37, 4], and Splay [21]. P2 utilizes the Overlog declarative language as an efficient way to specify overlay networks. Its data-flow design lends to effective parallelization, but its performance is not optimized for data centers. Libasync, its parallelization companion, libasync-mp, and event language Flux, provide another highly optimized toolkit for running event-driven and parallel programs. But again, the focus is on distributed event processing with asynchrony, not its combination or ability to handle automated rollback-recovery. Splay's focus, beyond the basic language and runtime, focuses on deployment and fair resource sharing across applications, and does not target data center environments.

To target data centers, MaceKen focuses on adding reliability to common data center failure-restart conditions which are not the expected failure case in wide-area distributed systems. Additionally, MaceKen targets resource usage based on expected resource availability in emerging data centers—network bandwidth is assumed to be abundant, and non-volatile RAM is available for efficient local checkpointing, while still needing to be frugal with system memory.

Rollback Recovery Rollback-recovery protocols have a long history [9]. These protocols include both checkpoint-based and log-based protocols, which differ in whether they record the state of a system or log its inputs. Combinations of checkpoint- and log-based systems continue to be popular, such as Friday [13], which uses system logs on a distributed application to run a kind of gdb-like debugger. A major challenge in rollback-recovery systems is to be able to rollback or replay state efficiently across an asynchronously connected set of nodes; in addition to checkpointing overhead, *coordination* overhead can be significant both in failure-free operation and during recovery. Moreover, systems that prevent failures from altering distributed computations in any way at all are overkill for a broad class of distributed systems that require only the weaker guarantee of output validity. Accepting this weaker guarantee actually provides Ken two distinct advantages: first, output validity can be preserved with a simple coordination-free local protocol, and second, it can actually allow a system to survive in some cases when the original sequence of events would lead to a persistent failure. See Lowell et al. for a detailed discussion of how and to what extent nondeterminism helps systems like Ken to recover from failures [25]. Ken differs from the well-known folklore approach of checkpointing atomically with every message/output because Ken bundles messages and outputs into turns, which simplifies the implementation and provides transactional turns that facilitate reasoning about distributed event-driven computations. Correia et al. execute event handlers on multiple local state copies to detect and contain arbitrary state corruption [7]. This complements Ken's crash-resilience handling by automating corruption checks.

7 Conclusions

The Ken rollback-recovery protocol protects local process state from crash/restart failures, ensures pairwise-FIFO message delivery and exactly-once message processing, and provides strong global correctness guarantees for distributed computations—distributed consistency and output validity. Ken's reliability guarantees furthermore *compose* when independently developed distributed systems interact or merge. Ken complements high-level distributed systems toolkits such as Mace, which raise the level of abstraction on asynchronous event-driven programming. Our integration of Ken into Mace simplifies Mace programs and enables them to adapt to new managed environments prone to correlated failures. Our tests show that our integrated MaceKen toolkit is versatile enough to tackle distributed programming problems ranging from graph analyses to DHTs, providing crash resilience across the board.

Acknowledgments

We would like to thank our shepherd, John Regehr, and our anonymous reviewers for their helpful comments. Research support is provided in part through the HP Labs Open Innovation Research Program. This research was partially supported by the Department of Energy under Award Number DE–SC0005026. See `http://www.hpl.hp.com/DoE-Disclaimer.html` for additional information.

References

[1] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing*, 1999. doi:10.1109/FTCS.1999.781058.

[2] R. J. Anderson. Why cryptosystems fail. *Commun. ACM*, 37, Nov. 1994. doi:10.1145/188280.188291.

[3] S. Ard and T. Clark. eBay blacks out yet again, June 1999. http://news.cnet.com/eBay-blacks-out-yet-again/2100-1017_3-226987.html.

[4] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A language for programming high-performance servers. In *USENIX ATC*, 2006.

[5] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM TOCS*, 3(1):63–75, Feb. 1985. doi:10.1145/214451.214456.

[6] T. Close. Waterken, 2009. http://waterken.org/.

[7] M. Correia, D. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *USENIX ATC*, 2012.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004. acmid:1251264.

[9] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, Sept. 2002. doi:10.1145/568522.568525.

[10] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *ACM CCS*, 2008. acmid:1455793.

[11] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[12] V. K. Garg. *Elements of Distributed Computing*. Wiley, 2002.

[13] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *NSDI*, 2007. http://www.usenix.org/event/nsdi07/tech/geels.html.

[14] The Graph500 Benchmark. http://www.graph500.org/.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS OS Rev.*, 41:59–72, Mar. 2007. acmid:1273005.

[16] T. Kelly. http://ai.eecs.umich.edu/~tpkelly/Ken/.

[17] T. Kelly, A. H. Karp, M. Stiegler, T. Close, and H. K. Cho. Output-valid rollback-recovery. Technical report, HP Labs, 2010. http://www.hpl.hp.com/techreports/2010/HPL-2010-155.pdf.

[18] C. Killian. http://www.macesystems.org/maceken/.

[19] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *FSE*, 2010. doi:10.1145/1882291.1882297.

[20] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *PLDI*, 2007. doi:10.1145/1250734.1250755.

[21] L. Leonini, É. Rivière, and P. Felber. Splay: Distributed systems evaluation made simple. In *NSDI*, 2009. Available from: http://www.usenix.org/event/nsdi09/tech/.

[22] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. In *ICDCS*, volume 43, 1988.

[23] S. Lohr. Smart dust? Not quite, but we're getting there. *New York Times*, Jan. 2010. http://www.nytimes.com/2010/01/31/business/31unboxed.html.

[24] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005. doi:10.1145/1095810.1095818.

[25] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *OSDI*, 2000.

[26] lxc Linux containers. http://lxc.sourceforge.net/.

[27] D. Mazières. A toolkit for user-level file systems. In *USENIX ATC*, 2001.

[28] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011. http://www.usenix.org/event/nsdi11/tech/full_papers/Murray.pdf.

[29] T. Negrino and D. Smith. *JavaScript and AJAX*. Peachpit Press, seventh edition, 2009.

[30] http://linux-mm.org/OverCommitAccounting.

[31] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM Press, 2000.

[32] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX ATC*, 2004. http://www.usenix.org/event/usenix04/tech/general/rhea.html.

[33] I. Steiner. eBay blames search outage on listings surge, Nov. 2009. http://www.auctionbytes.com/cab/abn/y09/m11/i21/s02.

[34] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI*, 2002. http://www.usenix.org/event/osdi02/tech/vahdat.html.

[35] H. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *SOSP*, 2007.

[36] S. Yoo, H. Lee, C. Killian, and M. Kulkarni. Incontext: simple parallelism for distributed applications. In *HPDC*, 2011. doi:10.1145/1996130.1996144.

[37] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX ATC*, June 2003.