Output-Valid Rollback-Recovery

Terence Kelly, Alan H. Karp, Marc Stiegler, Tyler Close, Hyoun Kyu Cho

HP Laboratories
HPL-2010-155

Keyword(s):

fault tolerance, rollback recovery, distributed computing, Ken, Waterken, checkpointing, nonvolatile memory.

Abstract:

Previous research has proposed and analyzed a broad spectrum of rollback-recovery protocols for fault-tolerant distributed computing, some of which are frequently used in specialized domains such as scientific computing. Adoption has been less widespread in commercial software development, however, because pragmatic engineering concerns and business requirements weigh against the properties of some existing protocols. Furthermore, a confluence of technology trends is casting doubt on some popular approaches to rollback-recovery while breathing new life into techniques less well suited to older technologies.

This paper analyzes an application-transparent rollback-recovery protocol for crash/recover hosts and fair-loss links. The protocol, Ken, is abstracted from an open-source implementation, Waterken, designed to facilitate reliable distributed commercial application development. Ken unifies application state checkpointing with logging required for reliable communication and is well suited to current technology and to the requirements of decentralized commercial software development. It preserves the main advantages of pessimistic logging, including simple local recovery and the need to maintain only one checkpoint per process. However it relaxes the very strong correctness guarantee provided by previous log-based approaches to avoid the increasing computational cost of providing it on current and foreseeable hardware. Ken provides a weaker yet still satisfactory guarantee, output validity: even in the presence of failures, the outside world sees outputs that could have resulted from failure-free operation.

External Posting Date: October 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: October 21, 2010 [Fulltext]

Output-Valid Rollback-Recovery

Terence Kelly[1] Alan H. Karp[1] Marc Stiegler[1] Tyler Close[2] Hyoun Kyu Cho[1,3]

[1]Hewlett-Packard Labs [2]Google [3]U. Michigan EECS

{terence.p.kelly, alan.karp, marc.d.stiegler}@hp.com
tyler.close@gmail.com netforce@eecs.umich.edu

September 23, 2010

Abstract

Previous research has proposed and analyzed a broad spectrum of rollback-recovery protocols for fault-tolerant distributed computing, some of which are frequently used in specialized domains such as scientif c computing. Adoption has been less widespread in commercial software development, however, because pragmatic engineering concerns and business requirements weigh against the properties of some existing protocols. Furthermore, a conf uence of technology trends is casting doubt on some popular approaches to rollback-recovery while breathing new life into techniques less well suited to older technologies.

This paper analyzes an application-transparent rollback-recovery protocol for crash/recover hosts and fair-loss links. The protocol, Ken, is abstracted from an open-source implementation, Waterken, designed to facilitate reliable distributed commercial application development. Ken unif es application state checkpointing with logging required for reliable communication and is well suited to current technology and to the requirements of decentralized commercial software development. It preserves the main advantages of pessimistic logging, including simple local recovery and the need to maintain only one checkpoint per process. However it relaxes the very strong correctness guarantee provided by previous log-based approaches to avoid the increasing computational cost of providing it on current and foreseeable hardware. Ken provides a weaker yet still satisfactory guarantee, *output validity*: even in the presence of failures, the outside world sees outputs that could have resulted from failure-free operation.

1 Introduction

Rollback-recovery protocols allow distributed computations in asynchronous message-passing systems to tolerate hardware and software failures by resuming execution after recovering pre-failure state from reliable storage. Thirty years of research have yielded a rich variety of well-understood, widely known techniques [12] that are sometimes used in specialized domains such as high-performance scientif c computing [11], where they help long computations run to completion. Despite the considerable theoretical attractions of rollback-recovery protocols, however, adoption has been limited in general distributed computing contexts such as commercial software development. To identify and overcome impediments to broader adoption we must examine relationships among the properties of such protocols, the capabilities of available and emerging hardware technologies, and the pragmatic business requirements and engineering concerns surrounding software development.

This paper analyzes a rollback-recovery protocol, Ken, abstracted from its implementation in an open-source software platform, Waterken, intended for commercial application development [37]. Building a reliable distributed peer-to-peer f le sharing application atop Waterken taught us to appreciate the practical benef ts of Ken [31]. Ken incorporates well-known elements that, to the best of our knowledge, have not been combined in the same way in prior literature. We believe that it deserves wider recognition because it is well suited to the requirements of distributed application development in several commercially important domains, e.g., next-generation worldwide sensor/actuator systems such as CeNSE [22] and massive Internet data centers serving an enormous worldwide user base via mobile clients. This paper formally describes Ken, characterizes its properties, and compares it with existing approaches in light of current trends in technology and application development.

As the theory surrounding rollback-recovery has matured, a growing recognition has emerged that it must adequately address practical considerations to achieve its full potential for real-world impact. For example, whereas early literature emphasized formal properties such as consistency among checkpoints, later research gave equal emphasis to software engineering issues such as application transparency, because experience had shown that integrating checkpointing into real software is remarkably diff cult [3]. More recent contributions foreground the importance of judiciously layered software architectures to implement formal protocols [15]. In a similar vein, we emphasize the ways in which protocols advance or impede the agendas of software developers, operators, and users. Specif cally, we consider whether rollback-recovery protocols facilitate defensive programming during development, division of responsibility in operation, and forgiving recovery following failures.

We show that Ken shares very desirable practical properties with pessimistic log-based rollback recovery: It maintains only a single checkpoint per process, and recovery and output commit are purely local. However Ken does not rely upon the piecewise determinism assumption that allows log-based approaches to replay execution exactly during recovery. One practical benef t is that Ken avoids the need for logging determinants of local nondeterministic events during failure-free operation, a formidable performance overhead for today's typical software and hardware. Another benef t is that Ken permits software to avoid repeating failures following recovery by relaxing the conventional requirement that, after rollback, application state must evolve exactly as it did prior to the failure. Nonetheless, Ken still provides a thoroughly acceptable correctness guarantee: The outside world sees outputs that *could have* resulted from failure-free execution.

The remainder of this paper is organized as follows: We begin in Section 2 by describing hardware technology trends that create new opportunities, and new challenges, for rollback-recovery protocols. We also review the special demands of important new distributed computing environments where such protocols are badly needed. Existing rollback-recovery approaches were not designed with current technologies and requirements in mind, and Section 3 explains how they do not fully exploit the former or fully satisfy the latter. Section 4 describes our system and failure models, Section 5 presents the Ken protocol, and Section 6 analyzes its properties. We conclude in Section 7 by explaining how Ken plays to the strengths of some emerging technologies while sidestepping obstacles erected by others, and why it is well suited to the technical and social realities of decentralized software development in some of today's most economically important distributed computing contexts.

2 New Technologies and Requirements

Two hardware technology trends promise to invert conventional wisdom on rollback-recovery: durable synchronous writes to modestly priced storage devices are becoming very fast, and the failure-free overhead required for deterministic replay of general application software is becoming unacceptable.

It has long been known that fast solid-state storage can improve failure-free performance in checkpoint-based rollback-recovery [4]. Until recently, however, most implementations employed spinning disks requiring several milliseconds to perform writes—in the best case, roughly 4 ms corresponding to rotational delay in today's fastest disks [9]. Research literature often ref ected the stated or implicit assumption that synchronous checkpoints are inherently slow and cannot be performed frequently without adversely affecting performance.

Today, however, f ash storage can perform synchronous durable writes in a fraction of a millisecond [8] and pricier RAM-based non-volatile storage can write in under 15 microseconds [33]. *Sub-microsecond* writes will be available in the near future from inexpensive non-volatile PCRAM storage [21]. Indeed, leading computer architects expect large, fast, and cheap PCRAM to supplant volatile DRAM as *primary memory* in general-purpose computers due to its superior scalability [21, 28, 40]. PCRAM offers the possibility of checkpointing far more frequently than is possible with disk-based storage [10]; other emerging non-volatile memory technologies promise comparable benef ts.

A downside of several solid-state storage technologies is limited write endurance. Storage locations in f ash devices, for example, fail after a relatively small number of writes [8]. The implication for checkpointing is that writes should be kept small: For a f xed checkpoint rate, a reduction in the average size of writes yields a proportionate increase in the useful lifetime of a f nite-endurance storage device by reducing the rate at which checkpoint data overwrite storage locations. Finally, f ash requires relatively slow erasure of storage locations before they can be rewritten, which favors protocols that quickly obsolete all but the most recent checkpoint so they can be garbage collected and erased asynchronously. In summary, whereas spinning disks strongly discourage frequent synchronous checkpoints, current and anticipated solid state storage technologies readily support them, especially if they are small and eagerly recycled.

The shift to multicore processors is another hardware trend with important implications for rollback-recovery. For the foreseeable future, successive processor generations will offer more cores rather than faster cores. Only parallel application software can exploit the full performance potential of multicore chips, and multithreaded software running on multicore hardware is rapidly becoming the norm. The diff culty of debugging such software has motivated intense active research on its deterministic replay, which is also required to implement log-based rollback-recovery protocols. Unfortunately, it appears that deterministic replay of multithreaded software on multicore hardware entails logging overheads during failure-free operation that scale very poorly with core count [24]. Specialized hardware may reduce failure-free overhead, but such hardware is neither available nor expected in the foreseeable future.

Important practical requirements for rollback-recovery protocols arise from the special characteristics and sheer scale of emerging distributed computing contexts that demand fault tolerance. Mobile computers frequently interact with users and routinely lose/regain network connectivity. When connected they do not always enjoy high-bandwidth or low-latency communications; network latency for a message may exceed the time required to write the same message to durable local storage. Mobile devices also routinely crash due to battery exhaustion and resume service after recharging. Crashes are becoming more common even in carefully controlled environments such as enterprise data centers due to the increasing use of commodity hardware and operating systems [11]. Furthermore, computing environments are increasing in scale, geographic reach, interactivity, and dynamism, as exemplif ed by the "warehouse-sized computers" intimately linked to worldwide client browsers in leading-edge Internet companies [5]. In contrast to traditional scientif c computing centers where rollback-recovery has proven its value, newer environments encompass far more nodes, exchange messages over far greater distances, acquire and lose nodes routinely, and interact with the outside world far more frequently. Whereas simply re-starting a failed job (or failed constituent sub-tasks) is often a reasonable way to deal with failure in batch scientif c/analytic computations, this isn't an option for computations that continuously receive inputs and continuously emit outputs. To be practical, a rollback-recovery protocol must assume that failure/recovery is the norm, adapt to message latencies that may exceed local storage write latencies, deal with dynamic arrival and departure of nodes, and handle frequent external interactions.

Decentralized development and deployment of loosely coupled applications imposes further requirements on rollback-recovery protocols. Many of today's most commercially important distributed applications consist of components developed by independent teams and operated across multiple administrative domains. Ubiquitous "mashups," for example, typically integrate software and data from two independent Web sites on a client machine owned and operated by a third party [34]. Decentralized development can succeed only if components interact via well-def ned interfaces and their developers respect clear divisions of responsibility. The essence of decentralized development is that the software underlying distributed processes is not under unif ed control. The developers of one local component rarely know with certainty whether to regard other components as the outside world or the "inside world"—i.e., as other protocol-compliant processes. Practical rollback-recovery techniques must be easy for the average programmer to apply, avoid reliance on global knowledge, and avoid global cooperation during development or operation as much as possible. They must also *compose* across organizational boundaries spanned by applications, and must respect the attitudes that accompany such boundaries. Finally, they must allow local developers to adopt a prudent, conservative, defensive posture toward their counterparts in the larger system.

The implications for rollback-recovery protocols are straightforward: Application-transparent methods are preferable to those requiring developers to manage checkpointing/logging manually. It is impractical to f x the number of processes participating in an application or to require global knowledge of this number, or to f x network topology. Most importantly, the sociology of decentralized development strongly favors protocols that focus responsibility rather than diffusing it, that contain damage rather than propagating it, and that compose across administrative boundaries. Allowing one process to negatively impact others leads to f nger-pointing at run time, so it is undesirable if a crashed process can compel rollbacks in other processes. Similarly, while a sluggish process on the *application-level* critical path will inevitably slow the application, it is undesirable if purely *protocol-level* interactions allow a slow process to throttle the outputs of other processes. A protocol is unacceptable if it creates a "distributed system" in the sense of Lamport's humorous def nition [20].

Fortunately, pragmatic considerations create opportunities as well as challenges. For example, a purely theoretical treatment of rollback-recovery may assume a reliable underlying communications medium or may expose unreliable communications to higher layers. In practice, neither option is acceptable. Real networks drop packets, so reliable messaging must be added somewhere in the software stack. This requirement invites us to consider *integrating* communication reliability with application state rollback-recovery while keeping both transparent. Another opportunity arises from the relatively relaxed correctness requirement that prevails among practitioners: Failure-free output is acceptable by def nition, and a failure has been successfully masked so long as outputs *could have been* emitted by a failure-free execution. During recovery we therefore have a useful degree of freedom: we may avoid repeating the failure even if doing so alters the output (within reasonable bounds).

3 Related Work

Standard approaches to application-transparent rollback-recovery in asynchronous distributed computing contexts are systematically documented in a comprehensive survey article [12] and several distributed computing texts [13, 15, 19]. Agbaria & Friedman comparatively evaluate nine rollback-recovery protocols in the context of a stochastic model, assuming reliable underlying message transport [1]. This section summarizes known properties of several classes of protocols for asynchronous systems in light of the requirements of Section 2 and describes the history of Waterken. After Sections 5 and 6 present and analyze the Ken protocol in detail, Section 7 contrasts Ken with several additional approaches from the prior literature and summarizes Ken's advantages.

In uncoordinated protocols, processes checkpoint state independently. Recovery, however, requires coordination to identify a globally consistent set of per-process checkpoints; in the worst case, all processes must roll back to their initial state—the "domino effect." Failure-free operation may generate useless checkpoints that are not part of any globally consistent recovery line, and in the worst case each process must on average retain a number of checkpoints proportional to the number of processes in the system [36]. Uncoordinated protocols are poorly suited to our needs.

Coordinated checkpointing protocols lie at the opposite extreme. Simple variants employ a global snapshot algorithm such as Chandy-Lamport [7] to identify a globally consistent state and orchestrate corresponding per-process checkpoints; optimizations attempt, e.g., to minimize the number of processes that must participate in a checkpoint [18]. Coordination avoids useless checkpoints by ensuring that every per-process checkpoint is part of a globally consistent recovery line. One problem is that coordination time scales with the diameter of the communication network [32], which can lead to slow checkpoints in some topologies. For example, a random network in a peer-to-peer application has $O(\log N)$ diameter, where N is the number of processes, and a grid or torus network has $O(\sqrt{N})$ diameter. The increasing scale of modern applications together with dramatic reductions in durable write latencies overturn the conclusion, common a decade ago, that coordination overhead is negligible in comparison to stable storage writes [12]. Slow coordination is furthermore problematic if outside-world interactions are frequent, because outputs cannot be released during checkpointing. The failure of a single node causes *global* rollback, and massive scale can make failures so frequent that coordinated rollback-recovery impedes forward progress rather than promoting it [11].

Communication-induced checkpointing (CIC) protocols piggyback data on interprocess messages that compel recipients to take checkpoints; processes may take purely local checkpoints in addition to these forced checkpoints. Modern CIC approaches build upon the theory of "ZigZag paths" [25] to avoid useless checkpoints. For example, Wang develops a sophisticated protocol based on the related concept of "rollback dependency trackability" [35]. This specif c proposal piggybacks $O(N)$ data on messages, potentially limiting scalability. A general shortcoming of several CIC protocols is that they require process to maintain multiple checkpoints and require coordination during both recovery and output commit.

Log-based rollback-recovery protocols augment application-state checkpoints by logging to stable storage all inf uences that potentially affect the course of process computation, including local sources of nondeterminism (e.g., OS scheduling decisions and inputs from CPU performance counters). During recovery, such protocols roll processes back to past checkpoints and use the logs to replay process execution *exactly*, which has the unfortunate practical consequence of ensuring the recurrence of certain classes of failures (e.g., those due improbable but fatal thread interleavings or message arrival orderings). In the context of rollback-recovery, *perfect* deterministic replay is arguably a bug, not a feature. On the positive side, pessimistic (synchronous) logging protocols have the very desirable property that recovery is simple and local: failed processes are re-started from their most recent checkpoint, and other processes are unaffected; output commit is also local. The major practical problem with all log-based protocols is that the failure-free overhead of logging for deterministic replay is becoming prohibitive for multithreaded software on multicore hardware (Section 2).

Crashes can destroy both application state and interprocess messages, e.g., messages buffered in volatile memory in the OS kernel pending transmission to a remote process or delivery to a local process. In practice today, to the extent that state loss and message loss are addressed at all, they are almost always addressed *separately* and *manually* at the *application* layer. In one common pattern, developers employ a transaction-processing database to manage durable application state and use reliable message queue middleware (e.g., [29, 38]) to ensure transport reliability in the face of host crashes. Opportunities for error abound as developers manually orchestrate numerous delicate interactions between the transactions that evolve application state and those that govern message exchanges; the slightest local mistake can easily cause global inconsistency in a distributed computation. At the same time, performance may suffer as two streams of persistent writes interleave, and possibly interfere, at the storage layer. We shall see that Ken relieves developers of responsibility for both application state consistency and message reliability, and coalesces persistent storage writes for these two purposes. We hypothesize that Ken may open new opportunities for software architectures that are currently server-centric primarily due to the diff culty of getting fault tolerance right without a central state repository. Specif cally we hypothesize that Ken may make it much easier to re-architect some such applications along peer-to-peer lines, thus potentially extending to them the adoption dynamics advantages, scalability, and decentralization benef ts of P2P architectures.

Waterken implements a superset of Ken as a Java library designed to facilitate reliable distributed application development [37]. Important aspects of Ken trace their lineage through Waterken to the E programming language [23]. The basic unit of computation in E is a *turn*, which corresponds to the receipt and complete processing of a message by a method whose invocation it triggers. Waterken extends E by making turns *transactional* using language-level orthogonal persistence [39] capabilities built upon standard Java object serialization primitives. Whereas E externalizes messages as they are sent during a turn, Waterken guarantees all-or-none message release at the end of a turn. Waterken takes highly selective application-state checkpoints at the end of each turn: only objects modif ed during the turn, and objects reachable from them via object references, are checkpointed. Furthermore, checkpoints are taken after processing method invocations return, which ensures that irrelevant transient application state changes (e.g., temporary stack or heap memory touched during a turn but de-allocated before the end of the turn) are excluded from checkpoints, keeping the latter small. By contrast, methods at the operating system or virtual machine monitor level that checkpoint process address spaces cannot easily distinguish valuable application state from irrelevant transient data and tend to include both in checkpoints.

To ensure message reliability, the Ken protocol logs process outputs to stable storage. Johnson & Zwaenepoel describe a sender-based message logging protocol that sidesteps synchronous storage operations, replacing them with *volatile* memory logging [17]. Their approach offers superior performance during failure-free operation on fast local-area networks if stable storage is based on slow spinning disks and if processes promptly fulf ll their protocol duties. However it assumes deterministic processes, tolerates only a single process failure, and allows a message *sender* to throttle a *recipient's* outputs. More generally, approaches that checkpoint process state remotely have a similar drawback: Checkpointing during failure-free operation and recovery following failures are both non-local, requiring distributed coordination and allowing protocol-level interactions to introduce performance and availability problems not inherent in application-level logic.

4 System Model and Failure Model

Figure 1 illustrates our system model. Computational hardware *nodes* host software *processes* that perform computations and communicate with other similar processes/nodes and with the outside world. The def ning

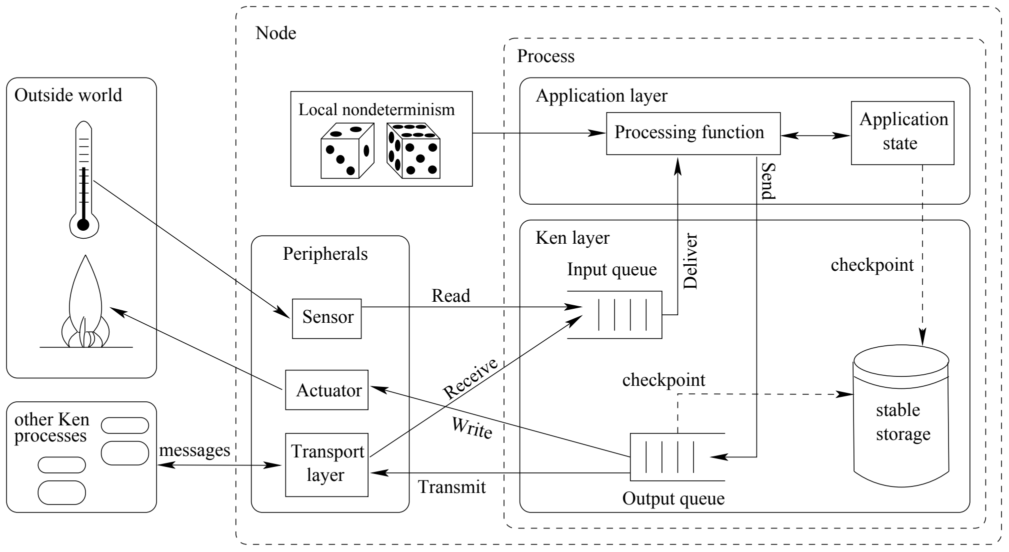


Figure 1: System model.

feature of the outside world is its inability to roll back or replay its interactions with the nodes under our control; tomorrow's weather will not warm us with yesterday's sunlight, and a rocket in f ight cannot be recalled to its launch pad. A process interacts with entities external to its node via peripherals: sensors and actuators that respectively *read* from the outside world and *write* to it, and a transport layer capable of *receiving* messages from other processes and *transmitting* messages to them. The system is asynchronous in the sense that we assume no bounds on message transit latency or on the relative progress of processes.

We divide processes into two layers. The Ken layer implements the rollback-recovery protocol of Section 5 and controls a *stable storage* device suitable for checkpointing and logging. The Ken layer also mediates interactions between application-layer software and entities external to the process by managing input and output queues and by employing peripherals. Application-layer software executes when the Ken layer *delivers* an input—a message from another process or data read from the outside world—to a *processing function*. The processing function may inspect and/or change application state and may *send* messages to other processes. Computations performed by the processing function may be affected by *local nondeterminism*, which models real-world phenomena including OS scheduling and branching on hardware performance counters or on the local clock. Scheduler artifacts include not only data races (which are disallowed in some languages) but also perfectly legal phenomena, e.g., branches that depend on which of several threads f rst acquires a mutex. Local nondeterminism may also introduce arbitrary physical-time delays into the processing function's progress; this models phenomena such as page faults. We make one further important assumption about local nondeterminism: it may not affect the state evolution of the processing function in ways that depend on past failure history, messages received, or outside-world inputs. So, for example, local nondeterminism may not allow the processing function to branch on the number of failures that the process has experienced to date.

A crucial property of processing functions is that they *terminate* in f nite time, i.e., during failure-free operation they return to the Ken layer that invoked them. A processing function is permitted to "fork" provided that all subordinate threads of execution thus created terminate prior to the return of the function. We say that a *turn* begins with the delivery of an input to a processing function and ends with the function's return. No messages are delivered during a turn other than the one whose delivery began the turn.

At arbitrary times processes may fail by *crashing*. A crash destroys all process state in volatile memory but does not affect the contents of the stable storage. Application state and the input and output queues in the Ken layer are lost during a crash unless they have been committed to stable storage, because they are maintained in volatile memory. Crashes model real-world failures of power, computer hardware, operating system software, language runtime systems, and application software. Causes of application software crashes include resource exhaustion and the inability of a processing function to handle an input (e.g., a buffer overf ow) or the particular local nondeterminism encountered during a turn (e.g., OS scheduler decisions). Following a crash, process *recovery* restores process state into volatile memory from data committed to stable storage prior to the crash, then resumes computation.

The transport layer may reorder messages in transit and may discard messages in "fair-loss" fashion [15]: every message will eventually be received at least once if it is transmitted suff ciently many times. For simplicity we do not consider the possibility that the transport layer may inject/fabricate or corrupt messages. Well-known techniques can effectively mask such failures. The same is true of dropped messages, of course, but we shall be interested in how Ken integrates message loss countermeasures with process crash remediation.

We assume that sensors reliably record inputs from the outside world at all times, even when their enclosing nodes crash, and retain inputs until the latter have been processed. Input devices ranging from tape drives to scientif c measurement instrumentation satisfy this assumption. Keyboards do not, but in practice few end-users continue typing into a crashed computer; instead they re-transmit their inputs following a reboot. We assume that actuators detect duplicate writes (e.g., if they occur following recovery) and ignore all but the f rst instance of a write. For this purpose it suff ces to associate a sequence number with each write and ensure that the actuator can maintain the sequence number of the most recent write across failures.

Given an initial state and inputs read from the outside world, a system of nodes writes a sequence of outputs to the outside world. We say that the latter is *valid* if it could have resulted from failure-free system operation. Our goal is to guarantee valid output even in the presence of process crashes and message loss.

5 The Ken Protocol

Figure 2 presents the Ken protocol in pseudocode for a main loop, an output loop, a peripheral handler (`receive()`), and a `send()` procedure available to application-level software. The main loop and the output loop run concurrently, i.e., in an implementation they may run in separate execution threads. The peripheral handler `receive()` asynchronously processes inputs from the transport layer; an analogous procedure, `read()`, handles sensors in a similar way and is not shown. Procedure `persist()` atomically commits specif ed volatile data structures to stable storage and `restore()` performs the inverse operation.

Outputs and messages sent by the processing function during a turn are buffered in the output queue until the end of the turn, when they are checkpointed to stable storage along with application state; only then are they transmitted/written and thus visible outside the process. Items are removed from the output queue only after they are reliably written to an actuator (in the case of outputs to the outside world) or acknowledged from recipients (in the case of messages). Outgoing messages are retransmitted until they are acknowledged, ensuring that they are delivered at least once. The `Done` table records messages that have been processed to completion. It is used to ensure that a message is delivered to the application at most once. It is also used to ensure that messages are delivered to the application layer in process-pairwise FIFO order (enforced by the `next_ready()` function, not shown, which straightforwardly considers messages' sequence numbers and sender IDs). A message that has already been processed to completion may be received again (e.g., due to a lost ack); it is simply acknowledged again but is not delivered to the application a second time. In summary, Ken provides applications with reliable "f re-and-forget" unidirectional datagrams guaranteed to be delivered exactly once in process-pairwise FIFO order.

At any moment, stable storage holds the most recently committed versions of the application state, output queue, `Done` table, and `turn` variable. Recovery consists of simply restoring these data structures into

```
global variables:                            // stored in volatile memory
    app_state:    array;                     // manipulated by processing_function()
    turn:         integer, initially zero;
    Q_in, Q_out:  data queues;
    Done:         table;                     // used by main() to ensure that each msg delivered
                                             //     at most once and by next_ready() to enforce
                                             //     process-pairwise FIFO delivery

start_or_recover() {
    restore(turn, app_state, Q_out, Done);
    spawn main();
    spawn output();
}

send(item: message, ack, or output data) {   // called by processing_function() and main()
    append pair (item, turn) to Q_out;       // buffer output for transmit() in output()
}

main() {                                     // runs concurrently with output()
    repeat forever {
        M <- next_ready(Q_in);               // blocks if Q_in empty or none ready according to Done
        if (M not completed in Done) {
            processing_function(M);          // deliver to application
            atomically {                     // checkpoint at end of turn
                turn <- turn + 1;
                record M completed in Done;
                persist(turn, app_state, Q_out, Done);
            }
        }
        send(acknowledgment of M);           // to process or sensor whence M originated
    }
}

output() {                                   // runs concurrently with main()
    repeat forever {
        for each pair (item, t) in Q_out {
            if (t < turn) {
                if (item is a message) {
                    transmit(item);
                }
                else {                       // item is data to be written
                    write(item);             // assumed to be reliable
                    delete (item, t) from Q_out;
                }
            }
        }
    }
}

receive(item: message or ack) {              // asynchronous handler; read() is similar
    if (item is a message) {
        insert message into Q_in;
    }
    else {                                   // item is an acknowledgment
        delete ack’d message from Q_out;
    }
}
```

Figure 2: Ken protocol.

volatile memory. Following recovery, the input whose processing failed will eventually be processed again; it may experience different local nondeterminism the next time around. Furthermore when the input queue is reconstructed from sender/sensor retransmissions following crash and recovery, it may deliver inputs to the application layer in a different order than would have occurred in failure-free operation. Recovery does not ensure that either of the two sources of nondeterminism that affect a failed receiver—the interleaving of messages from multiple senders, and local nondeterminism—will be replayed exactly.

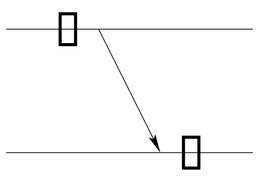
The most interesting aspect of the Ken protocol is how it treats different kinds of inputs. It logs interprocess messages at the sender's end to ensure reliable transmission to recipients in the face of message loss. However, unlike standard pessimistic log-based approaches, it simply ignores local nondeterminism. It ensures that each input is processed exactly once, but a crash during processing followed by recovery could result in application state evolving differently following recovery than it did prior to the crash. The transactional nature of turns ensures that failures are not visible outside the failed process. Intuitively, a crashed/recovered process is externally indistinguishable from a slow process, and this is the key to guaranteeing output validity.

The protocol of Figure 2 admits optimizations and enhancements that may be benef cial in an implementation. For example, messages in the input queue could be opportunistically persisted and acknowledged by recipients, allowing senders to garbage collect their output queues earlier. Incremental changes to application state and to the output queue may be persisted incrementally using well-known techniques. (Incremental checkpointing is compatible with the goal of rapidly garbage collecting obsolete checkpoints: A background job may coalesce incremental checkpoints asynchronously.) The `Done` table admits simple implementation; a table of most-recent sequence numbers for each communication partner in each direction suff ces. The Waterken implementation of Ken guarantees process-pairwise FIFO message delivery; stronger delivery ordering guarantees are possible. On a multicore computer it would be reasonable to run one Ken instance on each core to exploit the full potential parallelism of the machine, and communications between Ken instances on the same machine could be optimized to exploit shared memory and avoid network protocol overhead. We do not consider further such enhancements, implementation details, or performance optimizations.

6 Analysis

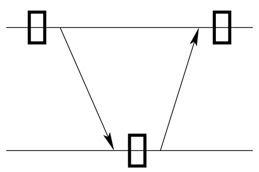
This section explains the most important properties of the Ken protocol: it takes no useless checkpoints; the set of most-recent per-node checkpoints is consistent; and the outside world sees only valid outputs even in the presence of tolerated failures.

We use *frontier* to refer to a set of checkpoints containing exactly one for each process. (The term "cut" is often used to refer to a set of per-process *states*; we use "frontier" to emphasize that the set includes only *checkpoints*, which may differ from process states, and also because we shall be interested in the most recent or "rightmost" checkpoint of each process.) A frontier is termed *consistent* if no checkpoint records the delivery of a message whose corresponding send is not recorded in some other checkpoint. A consistent frontier is also called a *recovery line*. A checkpoint is *useless* if it cannot be part of a recovery line. Inconsistencies arise when one checkpoint causally precedes another, in the sense of the happens-before relation, as illustrated in the process diagram at right. Horizontal lines represent processes, rectangles represent checkpoints, and arrows represent messages. Our system model includes four stages of message propagation: send, transmit, receive, and deliver. In our process diagrams, the tail of an arrow corresponds to *send* and the head corresponds to *delivery*. This is a subtle point. Recall that Ken buffers outgoing messages sent during a turn and externalizes the effects of the send only after committing the outgoing messages to stable storage as part of the end-of-turn checkpoint. The checkpoint ensures the



future transmission of messages sent during the previous turn; they will be transmitted by the `output()` procedure even if a crash destroys volatile memory. Therefore a message send is ref ected in the checkpoint at the end of the failure-free turn during which it occurs, even though the corresponding transmission cannot occur before the next turn.

A checkpoint is useless if and only if it is part of a cyclic ZigZag path, which generalizes the happens-before relation as follows [25]: A ZigZag path connects checkpoint C_1 in process A to C_2 in B if and only if the checkpoints are connected by a nonempty sequence of messages that originates in A after C_1 and terminates in B before C_2, and all pairs of consecutive messages (m_i, m_{i+1}) in the sequence have the property that m_{i+1} is sent by the same process that delivered m_i in the same or in a later checkpoint interval. The simplest example of a ZigZag cycle is illustrated at right. The cycle forms because the two messages are sent by and delivered to the upper process in the same checkpoint interval, and it renders the checkpoint in the lower process useless. It is easy to verify that a causal relationship exists between the checkpoint in the lower process and each checkpoint in the upper process, conf rming that the former is useless.

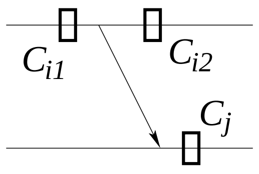


All ZigZag cycles have the feature illustrated in the upper process in the previous f gure: a send followed by a delivery in the same checkpoint interval [19]. It is easy to see that Ken precludes the formation of ZigZag cycles and thus takes no useless checkpoints. Checkpoint intervals correspond to turns, and each begins with the delivery of exactly one message followed by zero or more sends, with no further deliveries until the processing function returns and Ken takes its end-of-turn checkpoint, ending the checkpoint interval. A send followed by a delivery cannot occur in a checkpoint interval, and therefore ZigZag cycles cannot form and useless checkpoints are never taken. This observation follows from the constrained *pattern* of message delivery and sending that Ken enforces within checkpoint intervals. The *transactional* nature of turns allows us to establish a stronger result.

Theorem 1. *The set of most-recent per-process checkpoints is a consistent frontier.*

Proof. Assume otherwise. Then there is a pair of checkpoints C_i and C_j that are the most recent checkpoints in two different processes i and j and that are pairwise inconsistent, i.e., one of them, say, C_j, records the delivery of a message sent by process i but not recorded in checkpoint C_i. Because this message was delivered to process j, it must have f rst been received by j, and before that transmitted by process i. However because the message is not recorded in C_i, it must have been both sent and then transmitted by *i after* its most recent checkpoint C_i. This cannot occur because Ken's transactional turns ensure that the only messages externalized (transmitted) by a process are those that were sent *before* the most recent checkpoint. $\square$

Note that not every frontier is a recovery line; the above result applies only to the set of *most-recent* per-process checkpoints. Ken can give rise to the example at right, in which C_{i1} and C_j are not consistent. However the set of most-recent checkpoints, C_{i2} and C_j in this example, always constitutes a recovery line, and several important benef ts follow directly from this fact. Recovery consists of simply restarting crashed processes from their most recent local checkpoints. A crash in one process does not result in rollbacks in other processes, and the crash of one process during the recovery of another poses no additional diff culties. Crashes at arbitrary times in arbitrary numbers of processes cause local rollbacks to most-recent local checkpoints, and the resulting global system state can never become inconsistent.



We are concerned with practical systems that frequently interact with the outside world. One reassuring property is easy to establish: From our consistency result, and from the fact that Ken performs a local output commit in conjunction with every `write()`, we are assured that each *individual* output that reaches the outside world is ref ected in a consistent frontier (global checkpoint) of the system. Doubt may linger, however, about the effect that failures might have on the *collective* outputs reaching the outside world. Crashes destroy not only the message that a processing function is currently handling but also the input queue in volatile memory. Upon recovery, inputs may be delivered to a process in a different order than would have occurred in the absence of a crash, and the effect of each input on application state may be different due to the vagaries of local nondeterminism.

Our paramount concern is, informally, to ensure that failures do not alter the externally visible aggregate outputs of a system of processes in unacceptable ways. We say that a sequence of outputs to the outside world is *valid* if it could have resulted from failure-free execution, i.e., if it could have occurred in the absence of process crashes and message loss in transit.

Theorem 2. *Ken ensures that the sequence of outputs written to the outside world is valid.*

Output validity does not follow trivially from our consistency result; the latter is a property of sets of checkpoints whereas the former is a property of sequences of messages. Instead, output validity is a joint consequence of Ken's transactional turns and queue management. We will establish that recovery resets a process to a state that could have occurred in failure-free operation, and that subsequent inputs would be possible in failure-free operation. The net effect is that the observable behavior of processes does not betray the past occurrence of failures. Here we rely on our assumption that local nondeterminism does not provide a "covert channel" by which a process can infer the occurrence of past failures (Section 4).

Proof. A crash during checkpoint interval I destroys a process's volatile application state, input queue, and any messages sent since the start of I. No effects of computations during I prior to the crash are externally visible, because all messages and outputs sent in I are lost. Recovery yields a process state identical to that at the start of I, except that the input queue is empty. The same state could have occurred had the lost incoming messages been suff ciently delayed in transit instead of being destroyed by the crash, i.e., it could have resulted from failure-free operation. The subsequent observable behavior of the process is similarly indistinguishable from failure-free behavior because it depends only on subsequent message deliveries and local nondeterminism, and the future possibilities for these inf uences are not affected by the past crash. ☐

7 Discussion

In addition to the more sophisticated rollback-recovery protocols surveyed in Section 3, the prior literature contains several simpler approaches that do not piggyback detailed protocol data on messages and that rely on simple local rules to determine when to checkpoint application state. In our experience, observers sometimes confuse one of these earlier protocols with Ken, but in fact Ken differs in interesting ways. Bartlett describes a primary/backup co-processing scheme used to tolerate single failures in the Tandem NonStop operating system: Primary processes copy state to the volatile memory of their backup before every message transmission [6]. It is easy to show that useless checkpoints can result from a checkpoint-before-transmit protocol such as this (see, e.g., the second process diagram of Section 6). Ken persists outgoing messages prior to transmission, but constrains message delivery in such a way as to preclude useless checkpoints. Russell's MRS protocol forbids message delivery after any transmission within a checkpoint interval, and this bounds the extent of rollback required during recovery [30]. Ken allows multiple transmissions followed by a single delivery during each checkpoint interval, leading to the much stronger properties proven in Section 6. The CASBR protocol checkpoints after every transmission *and* before every delivery [35] but achieves no stronger guarantees than Ken, which checkpoints far less frequently and which is far more conducive to compact checkpoints. Checkpointing after every transmission [13] allows inconsistencies in the set of most-recent per-process checkpoints because an untimely crash in the sender can result in the most-recent checkpoints showing an un-sent message as having been delivered, so this protocol does not allow simple local recovery. Ken's transactional turns prevent inconsistencies and enable simple local recovery without the overheads of more complex techniques such as pessimistic logging. To the best of our knowledge Ken is not identical to any prior proposal, and it offers numerous attractions along several important dimensions:

Local Responsibility & Autonomy Ken localizes responsibility, contains damage, and facilitates defensive software development. Processes may wait for one another on the application-level critical path, but protocol-level interactions never allow one process to restrain another. Because the set of most recent per-process checkpoints is a recovery line, forward progress is ensured provided that every process can eventually digest every input. Processes may enter or leave the system dynamically—an increasingly important capability [2]—and neither the set of processes currently participating nor its cardinality nor the network topology must be known to any process for the protocol to work. Most importantly, Ken is well suited to decentralized software development and application deployment because it does not entangle recovery responsibilities. Simple local recovery is crucial to practical success for rollback-recovery protocols [16] and allows Ken to tolerate as many concurrent crashes as there are processes.

Composability & Adoption Dynamics Consider two independent systems of Ken processes, each exchanging messages internally but neither sending messages to the other. Ken guarantees output validity separately and independently for the two systems of processes. Now consider the consequences if message exchanges begin between the two systems: The *union* of the two systems becomes *collectively* output-valid, and this benef t accrues without any additional overt act on the part of the developers or operators of the two systems of processes. Systems of Ken processes compose effortlessly with respect to the output validity property. The implications for adoption dynamics are encouraging: The benef ts of collective output validity and the ease of achieving it amplify the incentives to adopt Ken.

Storage-Friendliness Ken plays to the strengths of modern storage devices, exploiting their ability to handle large numbers of small writes with low latency while making their lives easier in several ways. It stores no useless checkpoints and it coalesces application state checkpoints with outgoing message logging, minimizing writes to storage. It requires that each process maintain only one checkpoint, minimizing storage capacity requirements. It allows obsolete checkpoints to be garbage collected quickly, which helps f ash-based devices to erase reclaimed storage well before it is needed again. It helps implementations like Waterken to keep checkpoints small by writing them after processing functions have returned, making it easy to avoid writing transient stack, heap, and OS kernel data.

Interactivity The outside world is a f rst class citizen in several important classes of modern distributed applications that frequently interact with it. Ken's unif ed treatment of output commit and inter-process communication is appropriate when outside-world interactions are the common case.

Application-Transparency The event-driven style of programming that elementary Ken assumes is already ubiquitous: Programmers routinely structure software ranging from servers to user interfaces as collections of processing functions. Furthermore it is possible to build atop Ken programming patterns that programmers sometimes want while retaining all of Ken's benef ts. For example, Waterken integrates sophisticated "promise pipelining" capabilities inspired by the E language with the Ken rollback-recovery protocol [23, 37]. Our experience building and operating a reliable distributed peer-to-peer f le sharing system on Waterken convinces us that the combination of Ken's reliability guarantees and Waterken's high-level software facilities is remarkably convenient for developers [31].

Recoverability It has long been observed that most bugs in mature software are "Heisenbugs" whose failures do not recur during re-execution [14]. Ken affords recovering processes the opportunity to avoid repeating failures by exposing them to potentially different local and message-ordering nondeterminism than they saw before crashing. It thus may confer benef ts reminiscent of the Rx system, which *deliberately* alters a failed program's re-execution environment following rollback in the hope of preventing the failure from recurring [27]. The "second chance" that Ken offers is particularly helpful for crashes due to unlikely race conditions, an increasingly common cause of failure in the multicore era. Ken provides forgiving recovery by relaxing the conventional requirement of exact replay, instead allowing failures to cause different—but still valid—outputs to be emitted. Nightingale et al. exploit a similar relaxation to improve distributed f le system performance through speculative execution [26]. Their approach assumes unif ed system design, reducing the need for defensive programming and allowing better performance by obviating output commits for interactions among trusted components.

The only serious fundamental drawback of Ken is that it postpones process outputs until the synchronous end-of-turn checkpoint is complete. This can impair performance if processing function run times or storage write latencies are long relative to message transmission latencies, and if outside-world interactions are infrequent. Current trends in hardware (solid state storage), application type (interactive/commercial vs. batch/scientif c), and application scale (geographic distribution) are pushing in the opposite direction. We believe that Ken deserves consideration for the fault-tolerance needs of a wide array of emerging distributed computing applications in a broad range of contexts.

References

[1] A. Agbaria and R. Friedman. Model-based performance evaluation of distributed checkpointing protocols. *Performance Evaluation*, 65:345–365, 2008.

[2] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 17–25, Aug. 2009.

[3] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *Proc. Symposium on Fault-Tolerant Computing*, 1999.

[4] M. Banatre, G. Muller, and J.-P. Banatre. Ensuring data security and integrity with a fast stable storage. In *International Conference on Data Engineering*, pages 285–293, Feb. 1988.

[5] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[6] J. F. Bartlett. A NonStop Kernel. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 22–29, 1981.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of a distributed system. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[8] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of f ash memory based solid state drives. In *Proc. SIGMETRICS*, pages 181–192, June 2009.

[9] S. Chen. Flashlogging: Exploiting f ash devices for synchronous logging performance. In *Proc. SIGMOD*, pages 73–86, June 2009.

[10] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proc. International Conference for High Performance Computing, Networking, Storage, and Analysis*, Nov. 2009.

[11] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1(2):97–108, Apr. 2004.

[12] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

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

[14] J. Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, Nov. 1985.

[15] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2010.

[16] Y. Huang and Y.-M. Wang. Why optimistic message logging has not been used in telecommunications systems. In *Symposium on Fault-Tolerant Computing*, pages 459–463, 1995.

[17] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proceedings of the Symposium on Fault-Tolerant Computing*, 1987.

[18] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan. 1987.

[19] A. D. Kshemkalyany and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[20] L. Lamport, May 1987. "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt.

[21] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture (ISCA)*, pages 2–13, June 2009.

[22] 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.

[23] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, May 2006. http://www.erights.org/talks/thesis/index.html.

[24] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A software-hardware interface for practical deterministic multiprocessor replay. In *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 73–84, Mar. 2009.

[25] R. H. B. Netzer and J. Xu. Necessary and suff cient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, Feb. 1995.

[26] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed f le system. *ACM Transactions on Computer Systems*, 24(4):361–392, Nov. 2006.

[27] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3), 2007.

[28] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture (ISCA)*, pages 24–33, June 2009.

[29] Rabbit Message Queue, Feb. 2009. http://www.rabbitmq.com/.

[30] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, Mar. 1980.

[31] M. Stiegler. A reliable and secure application spanning multiple administrative domains. Technical Report HPL-2010-21, HP Labs, Feb. 2010. http://library.hp.com/techpubs/2010/HPL-2010-21.html.

[32] G. Tel. *Distributed Algorithms*. Cambridge University Press, second edition, 2000.

[33] Texas Memory Systems RamSan 400, Feb. 2009. http://www.ramsan.com/products/ramsan-400.htm.

[34] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Symposium on Operating Systems Principles (SOSP)*, pages 1–16, 2007.

[35] Y.-M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, Apr. 1997.

[36] Y.-M. Wang, P.-Y. Chung, and W. K. Fuchs. Tight upper bound on useful distributed system checkpoints. Technical report, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 2001. Earlier version available as UILU-ENG-95-2225, July 1995.

[37] Waterken, Feb. 2009. http://sourceforge.net/projects/waterken/.

[38] IBM WebSphere Message Queue, Feb. 2009. http://www-01.ibm.com/software/integration/wmq/.

[39] Wikipedia. http://en.wikipedia.org/wiki/Persistence_(computer_science).

[40] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy eff cient main memory using phase change memory technology. In *International Symposium on Computer Architecture (ISCA)*, pages 14–23, June 2009.