Killing the Chaos Monkey ... and Putting It ToWork

Marc Stiegler

HP Laboratories

HPL-2012-213

Abstract:
The Chaos Monkey test harness, which randomly kills nodes in a cluster, achieved notoriety after the Amazon cloud crashed on April 21, 2011 [8]. Many Amazon services experienced debilitating failures. But Netflix, having tested each software release with the Monkey before deployment, continued to operate effectively with mild degradation. Here we look at how the algorithms in the Clusterken framework – Ken protocol checkpointing, promise pipelining, domain redirection -- work together to render the Monkey obsolete. We then see how Clusterken can employ a tamed version of the Monkey to protect availability, and even facilitate load balancing.

External Posting Date: October 6, 2012 [Fulltext]
Internal Posting Date: October 6, 2012 [Fulltext]

Approved for External Publication

Killing the Chaos Monkey ... and Putting It To Work

Marc Stiegler
HP Labs

Abstract

The Chaos Monkey test harness, which randomly kills nodes in a cluster, achieved notoriety after the Amazon cloud crashed on April 21, 2011 [8]. Many Amazon services experienced debilitating failures. But Netflix, having tested each software release with the Monkey before deployment, continued to operate effectively with mild degradation. Here we look at how the algorithms in the Clusterken framework – Ken protocol checkpointing, promise pipelining, domain redirection — work together to render the Monkey obsolete. We then see how Clusterken can employ a tamed version of the Monkey to protect availability, and even facilitate load balancing.

1 Introduction

Distributed software applications, including both cluster-oriented applications and applications that span multiple administrative domains, bring a host of new threats to application reliability. A sample of simple problem scenarios that arise when node 1 sends a message to node 2 include:

1. If node 2 crashes before the message is processed, the message may be lost.
2. If node 1 crashes before the acknowledgement is received, the restarted node 1 may resend the message, causing node 2 to process the message twice.
3. If node 2 must ask node 1 for additional information before answering node 1's message, either both nodes may deadlock (if node 1 has blocked, waiting for node 2's answer) or node 1 may experience a data race and corrupt the data (if node 1 spawns another thread for the incoming message).

Such problems are nominally the easy ones to solve in a distributed system. Yet hundreds of services, including ones with significant programming resources such as Reddit and FourSquare, went dark during the Amazon cloud failure. Clearly, though these problems may be solved on paper, they have not been solved in practice.

2 Testing as a Solution

A testing facility such as the Chaos Monkey [4], which randomly kills nodes in a cluster, is a good first step. It imposes discipline, forcing the developers to deal with the risks early rather than late. But manually tuning each individual application component to survive the Monkey is tedious, error-prone (the Monkey may not catch all the risks), and expensive – sufficiently expensive that the temptation to forgo the testing, particularly in the face of a hard deadline, is often overpowering. Can we not build a solution to the Monkey into the programming infrastructure itself? If we could, then applications would be reliable by construction in the face of these failures. Every version of the application from the first prototype would be robust. An infrastructure solution would render the Monkey obsolete.

3 Exactly Once Message Processing as a Solution: Ken Protocol

A large piece of the problem can be solved using the Ken protocol, which guarantees exactly once message processing. The basic Ken algorithm has the following highlights [9]:

1. Before sending a message, the system checkpoints the node state and the new message together. The new message includes a unique id.
2. The message, including the unique id, is sent repeatedly until an acknowledgement (possibly containing a reply) is returned.

3. The recipient, upon receiving a message, checks the id. If this is a new message, the message is processed. Upon computing the answer, the system checkpoints the node state and the answer. Only after checkpointing is the message acknowledged (and the answer returned).

4. If the recipient receives a duplicate of the message (based on the unique id), it immediately returns the previously-computed answer.

All current implementations of Ken combine the protocol with transparent checkpointing and event driven, turn-based execution. A turn begins when a message is taken from the incoming event queue; the turn ends when the message has been processed to completion. All messages generated during the turn are recorded immediately, but are not released for transmission until after the turn has finished and the checkpoint for the turn has been taken.

Ken checkpoints are uncoordinated; no checkpoint is useless; each checkpoint advances the recovery line [6]. Recovering a node merely requires re-launching the checkpoint. When combined with programmer-transparent checkpointing, Ken meets some of the requirements that have been identified for future peta-scale systems [7].

At the application level, this algorithm guarantees that no crash-restart failure or sequence of crash-restart failures can prevent the application from proceeding from valid state to valid state. Programmers may write applications as if node failures and network partitions could not occur.

Consider the following example, a Java method built on a Ken-protocol framework, taken from a cluster-based, scalable publication/subscription system:

```
public Object receive(String event) {
    lastEvent = event;
    topic.addEvent(event, sender);
    return ok;
}
```

The topic object to which this receive method forwards the incoming event is on a remote node. Without Ken, receiver and topic would be susceptible to all the risks from network partitions and node crashes outlined earlier. The error handling code to ensure correct behavior of this three line method could easily be an order of magnitude greater than the code to do the specified job. But with Ken, the method becomes as simple as if it were part of a sequential program with the topic co-located with the receiver.

Ken-based reliability composes by default: if two independent systems are built using Ken, the meta system resulting from combining those two systems inherently has the same reliability properties [15]. This offers favorable adoption dynamics: given a few Ken-based systems, developers of systems that might want to integrate with those systems have an incentive to use Ken as well. This contrasts with global checkpointing, for example: two subsystems that use global checkpointing within themselves may still lack global reliability when brought into contact with one another.

The Ken protocol without any auxilliary features directly eliminates the risk from the first two example failure scenarios described earlier: messages cannot be lost or duplicated.

The simplest implementation of this algorithm is CKen, a distributed computing framework for C programmers. CKen has been tested with a simple Chaos Monkey, and it does indeed proceed to create valid output regardless of crashes.

CKen itself has been integrated with Mace to create MaceKen, a system that generates distributed applications using a domain specific language. Microbenchmarking indicates that Ken checkpoints can run as fast as ACID transactions in conventional databases.

To deal with the third example failure described earlier, CKen shares no state across threads (each application component resides in its own process), and uses send-and-forget messaging to avoid deadlock: to get a reply back, a component includes its own process as an argument in the message it sends.

4 Usable NonBlocking Messaging: Promise Pipelining

While programming with a send-and-forget messaging paradigm is possible, it can be difficult. A process can include itself in a message as the argument specifying where to send a result, but by the time the result comes in as a new message, the context for processing the result has been lost. Maintaining the context manually can yield code that is tedious to write, hard to read, and difficult to maintain. Early actors languages used this approach; none survive today. The most direct descendant of those languages to achieve success is Erlang, which has been used to build very reliable software systems [3]. However, one key to its success is the addition of a block-and-wait-for-answer mechanism, reintroducing the threat of deadlock. Indeed, books that teach Erlang often introduce block-for-answer immediately after introducing the safer send-and-forget semantics, and use it heavily in later examples [2]. In doing so, they acknowledge that send-and-forget is so cumbersome, it is better to risk introducing deadlock bugs in deployed systems during maintenance than to require programmers to consistently use this approach.

The Waterken Java framework [13, 5] for developing distributed applications not only implements the Ken protocol, it also implements promise pipelining [12, 10] for messaging. Briefly, when a message is sent, a promise for the eventual answer is immediately created on the sending side. Callbacks can be attached to the promise that will fire when the message recipient fulfills the promise. The protocol is nonblocking, hence avoids the risk of deadlock. And since the callback can be constructed as a closure (in Java, an inner class) that captures the state of the computation at the time of transmission of the message, it inherently has the context needed to utilize the reply when the reply eventually arrives. Promise callbacks can often be read and understood with little more difficulty than inline wait-for-answer code while still being safe.

5 Surviving Permanent Hardware Failures: the Redirectory

A Ken system cannot survive corruption of the stable store containing its checkpoints. Other mechanisms, such as hardware with RAID arrays, or software with replication and other more sophisticated algorithms, may be used to protect the checkpoints. But these mechanisms will run behind the scenes; the application programmer does not need to write special code to interface to them.

Other permanent hardware failures must be tolerated by the framework itself to fully survive the Chaos Monkey. If the hardware of one node fails, the checkpoints need to be launched on another node, possibly using a different IP address or even a different domain name. Hence Waterken supports a redirectory. When a Waterken server launches, it registers itself and its location with the redirectory (or redirectories, to avoid a central point of failure). The redirectory manipulates DNS to enable other servers to find the re-launched server whereever it may reside.

6 Surviving Failures Automatically: the Chaos Monkey

This combination of Ken protocol, promise pipelining, and redirecting still does not quite tolerate the full power of the Chaos Monkey. If a node goes down, the restarts of the checkpoints from that node must be automatic. Numerous algorithms exist for determining when a checkpoint should be relaunched: watching for unresponsive nodes, or inserting spies to achieve perfect failure detection [1] are samples. A simple alternative strategy with none of these features, that instead exploits the reliability features of Waterken, is now being investigated for Clusterken [14]. Clusterken is a layered framework built on top of Waterken for writing cluster-oriented applications. Clusterken has been used to demonstrate the advantages of relieving the programmer of the burden of programming error handling code for node and network failures. In comparing a Clusterken implementation of a publication/subscription system spec, to an implementation using Hadoop, the Clusterken version required only 1/4th as much code. Even more interesting from a dependability perspective, the Hadoop version would sometimes duplicate events when Hadoop restarted tasks; no such corruption was possible in the Clusterken version.

We are pursuing a Chaos-Monkey-derived strategy in Clusterken for automatic recovery that survives permanent hardware failure. Rather than attempting to detect a crashed node or a nonresponsive server, in Clusterken a tamed Chaos Monkey will, at regular intervals, move a server's checkpoint and relaunch the checkpoint on a different node. The original server, if it is still running, will crash when it tries to update its checkpoint and discovers that the checkpoint has ceased to exist (though the stable store may replicate checkpoint data for reliability, only a single server can run a checkpoint at a single time). Since the old server does not release its answer before it crashes, the sender of the last message will not get an answer; rather, it will detect the loss of its comm session, re-find the server's new instantiation on the new node via the redirectory, and then resend the message.

This algorithm requires neither leases nor synchronized clocks. It does not even require contacting the server: the server crashes upon discovery that its checkpoint is gone. Types of confusion such as exchanging messages with the wrong server are not possible: the checkpoints are associated with public/private key pairs, which are used to ensure one is connected to the correct server while establishing the comm session.

We currently estimate the time to move the checkpoint, relaunch, redirect, and reconstruct a comm session to be about 3 seconds. If one relaunched all the checkpoints over every ten minute period of computation, this would mean about 0.5% of the servers in a cluster would be out of action at any given moment. A further enhancement to the Monkey, to relaunch the checkpoint on an under-utilized node rather than a node selected at random, would extend its usefulness to load-balancing.

7 Surviving Programmers

Incorporating a tame Monkey would also continue to serve the original Chaos Monkey purpose of enforcing discipline. A programmer can breach the reliability of a Clusterken system by affecting the outside world via out-of-band channels. For example, if the application programmer appends directly to a java.io.File object, rather than using the Clusterken checkpointing system or the integrated Clusterken FarFile API for storing data, the append operation could create a duplicate entry. The continuing presence of a Monkey could detect such violations. Alternatively, the application development team could enforce this discipline by employing the Joe-E verifier [11] in conjunction with Clusterken. Joe-E was developed to enforce object-capability security discipline. All violations of the assumptions underpinning Clusterken reliability also violate object-capability security, since they require the use of channels that were not explicitly authorized. Hence any application that passes the verifier is guaranteed to follow the rules for reliability. Waterken and Clusterken were designed to cleanly support Joe-E verification. Indeed, Waterken itself is Joe-E verified.

8 Conclusion

When the tame Monkey implementing round-robin kill-move-relaunch is incorporated in Clusterken, the Chaos Monkey will be dead. However, it will have been reborn as a mechanism to protect availability. Long live the Chaos Monkey.

While building the Chaos Monkey, Netflix also wrote the maxim, "The best way to avoid failure is to fail constantly". Using a modified Chaos Monkey to continually crash servers by periodically moving each checkpoint's execution from one node to another takes this philosophy to a new level of intensity. When coupled with Ken protocol, promise pipelining, and redirectory services, it may enable the creation of highly reliable cluster-oriented programs that cost no more to build or maintain than the applications of questionable reliability found ubiquitously on the web today.

9 Acknowledgments

We would like to thank Alan Karp for his insights during a review of this paper.

10 Availability

CKen and MaceKen are being open-sourced under the BSD license. Further information on CKen can be found at

`http://ai.eecs.umich.edu/~tpkelly/Ken/`

Further information on MaceKen can be found at

`http://www.macesystems.org/maceken/`

Waterken is open-sourced under the MIT-X license. It is available at

`http://waterken.org`

Clusterken is being open-sourced under the LGPL license. Direct programmable access to a tiny Clusterken cluster can be acquired at

`http://www.skyhunter.com/pubshare`

References

[1] AGUILERA, M. K., AND WALFISH, M. No time for asynchrony. In *Usenix Workshop on Hot Topics in Operating Systems* (2009).

[2] ARMSTRONG, J. *Programming Erlang*. Pragmatic Programmers, 2007. ISBN-10:1-9343560-0-X.

[3] ARMSTRONG, J. What's all this fuss about erlang? *Pragmatic Bookshelf* (2007).

[4] CIANCUTTI, J. 5 lessons we've learned using aws. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html.

[5] CLOSE, T. Waterken server. http://waterken.sourceforge.net/.

[6] ELNOZAHY, E. M., ALVISI, L., MIN WANG, Y., AND JOHNSON, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys* (September 2002).

[7] ELNOZAHY, E. N., AND PLANK, J. S. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing* (June 2004).

[8] GILBERTSON, S. Lessons from a cloud failure: It's not amazon, it's you. http://www.webmonkey.com/2011/04/lessons-from-a-cloud-failure-its-not-amazon-its-you/.

[9] KELLY, T., KARP, A. H., STIEGLER, M., CLOSE, T., AND CHO, H. K. Output-valid rollback-recovery. Tech. Rep. 155, HP Labs, 2010.

[10] LISKOV, B., AND SHRIRA, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI* (1988).

[11] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-e: A security-oriented subset of java. In *Network and Distributed System Security Symposium* (2010).

[12] MILLER, M. S. *Robust Composition: Towards A Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins, 2006.

[13] STIEGLER, M. A reliable and secure application spanning multiple administrative domains. Tech. Rep. 21, HP Labs, 2010.

[14] STIEGLER, M., LI, J., KAMBATLA, K., AND KARP, A. Clusterken: A reliable object-based messaging framework to support data center processing. In *Open Cirrus Summit* (2011). Tech report version at http://www.hpl.hp.com/techreports/2011/HPL-2011-44.html.

[15] YOO, S., KILLIAN, C., KELLY, T., CHO, H. K., AND PLITE, S. Composable reliability for asynchronous systems. In *USENIX Annual Technology Conference* (2012).