Composable Reliability for Asynchronous Systems

Sunghwan Yoo, Charles Killian, Terence Kelly, Hyoun Kyu Cho, Steven Plite
Distributed systems: Key-value store

Distributed systems: Key-value store
Distributed systems: Key-value store

Retransmission
Restart upon crash-restart
Rollback-recovery protocol
- Checkpoint-based
- Message-logging based
Distributed systems: Handling failures
Distributed systems: Handling failures

Distributed systems: Handling failures

Guaranteeing global reliability across independently developed components is difficult.
Ken: Crash-restart tolerant protocol
Ken: Crash-restart tolerant protocol

Ken preserves global **reliability** when you **compose** independent components
Ken Highlights

1. Reliability
   - Uncoordinated rollback recovery protocol
   - It’s also scalable

2. Composability
   - Write locally, work globally

3. Easy Programmability
   - Event-driven system (not a new paradigm)
   - Transparently applicable to Mace

Ken makes crash-restarted node look like slow node
Related Work

- Rollback Recovery
  - Much research through 1990s
  - Waterken (1999) : Ken principles in different programming abstractions
  - Lowell et al. (2000) : Output validity
  - Computing Surveys (2002): summary of mature field

- Software Persistent Memory (ATC 2012)
  - Different approach to orthogonal persistence

- Hardening Crash-Tolerant Systems (ATC 2012)
  - Detects arbitrary state corruption in event-driven code
  - Could make Ken-based software more reliable
Design of Ken

- Communicating event loop

WORLD

KEN

KEN

KEN
Design of Ken

WORLD

KEN

EXTERNALIZER

Event loop begins

Changes to memory heap

Sending messages

Store as checkpoint file

Commit

Send out messages

handler()

Send out
messages

EXTERNALIZER

13

Crashed/recovered nodes look like slow nodes

Uncoordinated protocol → scalable

Acked msgs are removed
Ken: Composable Reliability

Seller

Buyer

Bank

k b a Y

WIN

26

16

Guaranteeing global reliability across independent components is a difficult task.
Ken: Composable Reliability

When you **compose** independent components, **reliability** will be transparently guaranteed by Ken

Ken allows **decentralized development**
Ken Illustrated: “Ping-pong Counter”

```c
#include "ken.h"
#include "kenapp.h"

int ken_handler(void *msg, int len, kenid_t src) {
    int *cnt;
    if (NULL == msg) {
        cnt = ken_malloc(sizeof *cnt);
        *cnt = 0;
        ken_set_app_data(cnt);
    }
    else {
        cnt = ken_get_app_data();
        *cnt = *(int*)msg + 1;
        ken_send(src, cnt, sizeof *cnt);
    }
    return -1;
}
```

- analogue of `main()`
- initialization
- begin transaction
- persistent heap
- entry point
- incoming message
- fire & forget
- end transaction
Ken Illustrated: “Ping-pong Counter”

```c
#include "ken.h"
#include "kenapp.h"

int ken_handler(void *msg, int len, kenid_t src) {
    int *cnt;
    if (NULL == msg) {
        cnt = ken_malloc(sizeof *cnt);
        *cnt = 0;
        ken_set_app_data(cnt);
    } else {
        cnt = ken_get_app_data();
        *cnt = *(int*)msg + 1;
        ken_send(src, cnt, sizeof *cnt);
    }
    return -1;
}
```

**Ken programming is simple**

1. Implement `ken_handler()` instead of `main()`
2. Use `ken_malloc / ken_send` instead of `malloc / send`
3. Use `ken_get_app_data / ken_set_app_data` for entry to persistent heap

**analogue of main()**

**fire & forget**

**end transaction**

Mace: “Ping-pong Counter”

```plaintext
Mace with Ken

```service PingPong;

```services { Transport t; }

```state_variables { int cnt = 0; }

```messages { pong {int cnt;} }

```transitions {
    deliver(src, dest, msg) {
        cnt = msg.cnt+1;
        route(src, pong(cnt));
    }
}

```

- Event-driven distributed system language framework
- Used in many projects

in persistent heap

define state var & messages

incoming message

begin transaction

send message

fire & forget messaging

end transaction
Mace: “Ping-pong Counter”

service PingPong;
services { Transport t; }

state

messages { pong { int cnt; } }

transitions {
  deliver(src, dest, msg) {
    cnt = msg.cnt + 1;
    route(src, pong(cnt));
  }
}

Mace with Ken

You don’t need to change anything for Ken

Reliability and composability comes easily

fire & forget messaging

end transaction

in persistent heap

No changes are needed
## Ken: Integration with Mace

<table>
<thead>
<tr>
<th>Feature</th>
<th>Ken</th>
<th>Mace</th>
<th>MaceKen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Masks failures globally</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Composable reliability</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Packaged with distributed protocols</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Availability through replication</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Handles permanent failures</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Ken provides new benefits to legacy Mace applications!**
Implementation

**Ken**
- C library
- Publicly available

**MaceKen**
- Modifications to existing Mace runtime libraries
- No changes to existing Mace services

**Linux Container (LXC) environment**
- Simulating correct power-failure behavior (in paper)
Evaluation

- Microbenchmark
  - Performance test

- kBay
  - Composable reliability (in paper)

- Distributed analysis of 1.1 TB graph
  - Versatility (in paper)

- Bamboo-DHT
  - Failure masking & data survival for legacy Mace app
Evaluation: Ken Microbenchmark

Experimental setup

- 16 core, 2.4 GHz Xeon
- 32GB RAM
- Mirrored RAID
  - Two 72 GB 15K RPM disks

Test

- Ping-pong counter test between two Ken processes

Measure: latency and throughput
Evaluation: Ken Microbenchmark

Latency (ms)

- disk sync
- no sync
- ramfs sync

Thruput (events/sec)

- 0
- 5,000
- 10,000
- 15,000
- 20,000
- 25,000
Evaluation: Bamboo-DHT

Wide-area network

Managed network

Single administration

Colocation

Evaluation: Bamboo-DHT

- Bamboo-DHT on Mace will fail
- Bamboo-DHT on MaceKen will work
- Mace vs. MaceKen: testing rolling-restart and correlated failure
Evaluation: Bamboo-DHT

Experimental setup
- 12 dual quad-core machines, 8GB RAM
- 4x1Gb Ethernet connections
- 300 DHT nodes on 12 machines

Measures
- Immediate-Get
  - X=1
  - X=1
  - X=1
- Stored-Get
  - X=1
  - X=1
  - X=1
Bamboo-DHT: Crash-restart Failures

Correlated Failures

100% data survivability
very low data survivability

begin experiment

Stored GET begins

100% data survivability

Mace immediate
Mace stored
MaceKen immediate
MaceKen stored

bootstrapped node spared

correlated failure
Bamboo-DHT: Rolling-restart

Rolling-restart

100% data survivability
still low data survivability

stored GET begins
churn begins
churn ends

success fraction

Time (min)

MaceKen provides automated crash resilience to legacy Mace app and 100% data survivability.
Conclusion

Ken provides
1. Reliability
2. Composability
3. Programmability

MaceKen provides
1. Transparency
2. 100% data survivability

Ken available now:
http://ai.eecs.umich.edu/~tpkelly/Ken/

MaceKen available soon:
http://www.macesystems.org/maceken/

Thank you!