Practical Hardening of Crash-Tolerant Systems

Marco Serafini (Yahoo! Research BCN)
Joint work with: Miguel Correia (IST-UTL / INESC-ID)
Daniel Gómez Ferro and Flavio Junqueira (Yahoo! Research BCN)
Dependability in data centers

0 Crashes are commonplace...
0 ... but scarier faults do occur
A horror story

An 8-hour system-wide outage due to a single hardware fault

Amazon S3 Availability Event: July 20, 2008

We wanted to provide some additional detail about the problem we experienced on Sunday, July 20th.

At 8:40am PDT, error rates in all Amazon S3 datacenters began to quickly climb and our alarms went off. By 8:50am PDT, error rates were significantly elevated and very few requests were completing successfully. By 8:55am PDT, we had multiple engineers engaged and investigating the issue. Our alarms pointed at problems processing customer requests in multiple places within the system and across multiple data centers. While we began investigating several possible causes, we tried to restore system health by taking several actions to reduce system load. We reduced system load in several stages, but it had no impact on restoring system health.
What happened?

Quoted from the Amazon service health dashboard:

- "A handful of messages had a single bit corrupted"
- "The message was still intelligible, but the system state information was incorrect"
- "We used MD5 checksums throughout the system (but not) for this particular internal state information"
- "(The corruption) spread throughout the system causing the symptoms described above"
Error propagation

Process $i$

$u$

$v$
Error propagation

Process $i$

$u$

$v$
Error propagation

$m_{in}$

Process $i$

$u$

$\mathbf{v}$
Error propagation

$m_{in}$

Process $i$

$u$

$v$

Event handling

$m_{out}$
Error propagation

$\text{Event handling}$

$\text{Process } i$

$m_{in}$

$u$

$\text{Event handling}$

$v$

$m_{out}$
Error propagation

Event handling

Process $i$

$m_{\text{in}}$

$u$

$v$

Process $j$

$m_{\text{out}}$

$x$

$y$

$m_{\text{in}}$
Error propagation

Event handling

Process $i$

$m_{in}$

$u$

$\rightarrow$

Event handling

$m_{out}$

$v$

Process $j$

$x$

$y$

$\rightarrow$

$m_{in}$
A new approach to error isolation

1. General model of process behavior
2. Arbitrary State Corruption (ASC) fault model
3. Guarantee error isolation through hardening
A new approach to error isolation

1. General model of process behavior
2. Arbitrary State Corruption (ASC) fault model
3. Guarantee error isolation through hardening
A new approach to error isolation

1. General model of process behavior
2. Arbitrary State Corruption (ASC) fault model
3. Guarantee error isolation through hardening
A new approach to error isolation

1. General model of process behavior
2. Arbitrary State Corruption (ASC) fault model
3. Guarantee error isolation through hardening
Data corruptions

0 Commodity disks are known to be unreliable
   0 Faulty firmware is the first reason
0 RAM: ECC errors are frequent
   0 Production machines only see detected errors
      → Coverage not known
0 Interconnects and CPUs also fail
   0 Faulty drivers or bit flips
Existing approaches

Background
Common practice

0 Manual placement of error detection checks
0 Application knowledge
0 Time consuming
0 Hard to structure without fault model
0 No error isolation guarantee
Byzantine fault model

0 **Black-box model** of faulty processes: adversarial
0 Hardening for error isolation [Nysiad NSDI 2008]
  0 Based on state machine replication
  0 Replication and performance costs
Byzantine faults

0 Byzantine hardening covers attacks and bugs...
0 ... assuming, e.g., design diversity of replicas
0 Unpractical in most systems
Byzantine faults

0 Byzantine hardening covers **attacks** and **bugs**...
0 ... assuming, e.g., **design diversity** of replicas
0 Unpractical in most systems
Process and fault models

Defining Arbitrary State Corruptions
Process model

1) Event Dispatching

2) Event Handling

3) Message sending

Upon receive message <REQ, r> do
if v > 5 then
  u = r + v + 5;
else
  u = r + v;
  v = u;
send <WRITE, v> to process p
An **Arbitrary State Corruption** can make a process
- Crash
- Assign an arbitrary value to any variable
- Start the execution from an arbitrary instruction

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>z</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>z</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>7</td>
<td>320</td>
</tr>
</tbody>
</table>
Fault frequency

0 One fault for every processed input message

1) Event Dispatching

2) Event Handling

3) Message sending

Upon receive message $<\text{REQ}, r>$ do
if $v > 5$ then
  $u = r + v + 5$;
else
  $u = r + v$;
$v = u$;
send $<\text{WRITE}, v>$ to process $p$
Fault diversity

0 A corrupted variable is different from its replica

0 Only holds immediately after the fault

0 Can be invalidated if instructions modify the variable
Error propagation

0 Fault diversity does not hold
0 Hardening preserves diversity

<table>
<thead>
<tr>
<th>Original</th>
<th>Replica</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$u$</td>
</tr>
<tr>
<td>$v$</td>
<td>$v$</td>
</tr>
</tbody>
</table>
Error propagation

0 Fault diversity does not hold
0 Hardening preserves diversity
Error propagation

0 Fault diversity does not hold
0 Hardening preserves diversity
Error propagation

0 Fault diversity does not hold
0 Hardening preserves diversity
ASC hardening

From ASC faults to crashes and message omissions
From ASC to crashes

0 **Transparent**: to the hardened process
0 **Local**: no process replication on multiple machines
0 **Untrusted**: can have faults while executing hardening
Simple hardening

\[
\begin{array}{c|c}
\text{Original} & \text{Replica} \\
\hline
u & \\
v & \\
\end{array}
\]
Simple hardening

\[
\begin{array}{c|c|c}
\text{Original} & \text{Fault diversity} & \text{Replica} \\
\hline
u & \quad & \\
\hline
v & \quad & \\
\end{array}
\]
Simple hardening

Event handling

Upon receive message <REQ, r> do
    if v > 5 then
        u = r + v + 5;
    else
        u = r + v;
    v = u;
    send <WRITE, v> to process p
Event handling

Upon receive message \(<REQ, r>\) do
  if \(v > 5\) then
    \(u = r + v + 5;\)
  else
    \(u = r + v;\)
  \(v = u;\)
  send \(<WRITE, v>\) to process \(p\)
Simple hardening

Upon receive message \(<\text{REQ}, r>\) do
if \(v > 5\) then
  \(u = r + v + 5;\)
else
  \(u = r + v;\)
\(v = u;\)
send \(<\text{WRITE}, v>\) to process \(p\)
Simple hardening

Upon receive message <REQ, r> do
  if v > 5 then
    u = r + v + 5;
  else
    u = r + v;
  v = u;
  send <WRITE, v> to process p

Error propagation!
Upon receive message \(<REQ, r>\) do
  
  if \(v > 5\) then
    \(u = r + v + 5;\)
  else
    \(u = r + v;\)
  \(v = u;\)
  send \(<WRITE, v>\) to process \(p\)
Protecting computation

Upon receive message <REQ, r> do
   if v > 5 then
      u = r + v + 5;
   else
      u = r + v;
   v = u;
   send <WRITE, v> to process p
Protecting computation

Upon receive message <REQ, r> do
  if v > 5 then
    u = r + v + 5;
  else
    u = r + v;
  v = u;
  send <WRITE, v> to process p
Checking the checkers

Event handling

Upon receive message <REQ, r> do
  if v > 5 then
    u = r + v + 5;
  else
    u = r + v;
  v = u;
  send <WRITE, v> to process p
Checking the checkers

Upon receive message <REQ, r> do
  if v > 5 then
    u = r + v + 5;
  else
    u = r + v;
  v = u;
  send <WRITE, v> to process p

Event handling
Checking the checkers

Event handling

Upon receive message <REQ, r> do
if v > 5 then
  u = r + v + 5;
else
  u = r + v;
  v = u;
send <WRITE, v> to process p
Checking the checkers

Upon receive message <REQ, r> do
  if v > 5 then
    u = r + v + 5;
  else
    u = r + v;
  v = u;
  send <WRITE, v> to process p
Issue with redundant checks

Original

$u$

$\nu$

Replica

$v$

$m_{in}$
Issue with redundant checks

Original

\( u \)

\( v \)

Replica

\( m_{out} \)

Event handling

\( m_{in} \)
Issue with redundant checks
Issue with redundant checks

Original

$u$

$\nu$

Event handling

Replica

$v$

Redundant event handling

$m_{out}$

$m_{in}$

CHECK
Incremental buffer

Original

Replica

Event handling

Redundant event handling

$m_{in}$

$m_{out}$
Control flow errors

0 A control flow error may subvert the execution
   0 An event handler could be executed twice
   0 Event handling may be skipped or incomplete

0 Requires control flow checks
   0 Use flags to control the control flow
   0 Very lightweight
PASC library

Process state

Replica state

EH₁

EH₂

EH₃

PASC runtime

PASC checks

User-defined

Transparent
PASC library

Process state

Replica state

EH₁
EH₂
EH₃

PASC checks

PASC runtime

User-defined

Transparent
PASC library

Process state

Replica state

PASC checks

EH₁
EH₂
EH₃

PASC runtime

User-defined

Transparent

github.com/yahoo/pasc
Evaluation
Hardening an echo server

PBFT
PASC Echo
Unprot. Echo

0 Little computation, network bound, no overhead
0 PBFT is a reference (Nysiad not available)
Hardening State Machine Replication

Latency in ms vs Throughput in Kops/s

- PBFT
- PASC Paxos
- Unprot. Paxos
Hardening State Machine Replication

- PBFT
- PASC Paxos
- Unprot. Paxos

Latency in ms vs Throughput in Kops/s graph showing
- +70% improvement for PBFT
- -15% improvement for Unprot. Paxos
Zookeeper

Latency in ms

Throughput in Kops/s

PASC ZooKeeper

Unprot. ZooKeeper
Memory overhead

![Bar chart showing memory usage in MB for different request sizes in bytes.
- Y-axis: Memory usage in MB
- X-axis: Request size in bytes (0, 1K, 4K)
- Bars represent Unprot. Paxos and PASC Paxos]

- Unprot. Paxos:
  - 0 bytes: 0 MB
  - 1K bytes: 1 MB
  - 4K bytes: 2 MB

- PASC Paxos:
  - 0 bytes: 2 MB
  - 1K bytes: 14 MB
  - 4K bytes: 14 MB

The chart illustrates the memory overhead for different request sizes, showing that PASC Paxos consumes significantly more memory compared to Unprot. Paxos.
Scalability

**SimpleKV**:
- Eventually consistent store, no replication
- Scales similarly with hardening
- No server “wasted” for replication

---

**Graph**

- X-axis: Number of servers
- Y-axis: Max. throughput (kops/sec)
- Line 1: PASC sKV
- Line 2: Unprot. sKV

---

0 SimpleKV: eventually consistent store, no replication
0 Scales similarly with hardening
0 No server “wasted” for replication
PASC fault coverage

0 Injected random bit flips in Paxos
  0 Code corruptions: bytecode and binary code
  0 State corruptions: pointers and primitive values

<table>
<thead>
<tr>
<th></th>
<th>Code corruptions</th>
<th>State corruptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unprot</td>
<td>PASC</td>
</tr>
<tr>
<td>Undet.</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Det.</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Crash</td>
<td>1640</td>
<td>1663</td>
</tr>
<tr>
<td>Total</td>
<td>2856</td>
<td>2856</td>
</tr>
</tbody>
</table>
Conclusions

0 Hardware data corruptions occur in data centers
0 Model as *Arbitrary State Corruptions*
0 ASC-hardening algorithm for *error isolation*
  - Local: does not require replication
0 PASC: ASC-hardening library
  - Efficient: PASC-Paxos has up to 70% more throughput than PBFT
  - High fault coverage
Thank you

serafini@yahoo-inc.com

github.com/yahoo/pasc