Fault-Tolerant Services in Distributed Systems Using

Vijay K. Garg

e-mail: garg@ece.utexas.edu
(includes joint work with Bharath Balasubramanian and V

ECE Dept., Univ. Texas at Austin
Modeling Services in Distributed Systems

- Server: a Deterministic State Machine: not necessarily finite
- Clients: Interact with Servers using events/messages
- Crash Fault: Server’s state is unavailable
- Byzantine Fault: Server’s state is corrupted
Example: Resource Allocation

\[\text{user}: \text{int initially 0};\]
\[\text{waiting}: \text{queue of int initially null};\]

On receiving acquire from client \textit{pid}

\[\text{if } (\text{user} == 0) \}\{\]
\[\text{send(OK) to client } \text{pid}; \text{user = pid;};\]
\[\text{else append(} \text{waiting, pid});\]

On receiving release

\[\text{if } (\text{waiting.isEmpty}());\]
\[\text{user} = 0;\]
\[\text{else } \{ \text{user} = \text{waiting.head}();\]
\[\text{send(OK) to user;}\]
\[\text{waiting.removeHead}(); \}\]
\( f \): maximum number of faults in the system

Crash faults: Keep identical \( f + 1 \) replicas of the server

- Use Determinism If an event applied, the resulting state is the same
- Agreement on the order Ensure that servers agree on the order of events

Byzantine faults: Keep identical \( 2f + 1 \) replicas of the server

- Use Voting If response is different, choose the response with the most votes
Our Setup

\( N \) different servers

Motivation:
- Multiple instances of state machine for different departments/stores/regions
- Partitioning the state machine for scalability

Replication
- Crash faults: \((f + 1)N\) states machines
- Byzantine faults: \((2f + 1)N\) states machines

Our Algorithms
- Crash faults: \(N + f\) states machines
- Byzantine faults: \((f + 1)N + f\) states machines
Event Counter Example, $f = 1$
$P(i) :: i = 1..n$

```plaintext
int count_i = 0;

On event entry(v):
  if (v == i) count_i = count_i + 1;
On event exit(v):
  if (v == i) count_i = count_i - 1;
```

$F(1) ::$

```plaintext
int fCount_1 = 0;

On event entry(i), for any i
  fCount_1 = fCount_1 + 1;
On event exit(i) for any i
  fCount_1 = fCount_1 - 1;
```

Figure 1: Fusion of Counter State Machines
Using Order in Distributed Computing

Issues

- Multiple faults
- More complex data structures
- Overflows
- Byzantine faults

ECE Dept., Univ. Texas at Austin
Multiple Faults

$$F(j) :: j = 1..f$$
$$\text{int } f\text{Count}_j = 0;$$

On event $entry(i)$, for any $i$
$$f\text{Count}_j = f\text{Count}_j + i^{j-1};$$

On event $exit(i)$ for any $i$
$$f\text{Count}_j = f\text{Count}_j - i^{j-1};$$

Figure 2: Fusion of Counter State Machines

$$f\text{Count}_2 = \sum_{i} i * \text{count}_i$$
\[ f\text{Count}_j = \sum_{i} i^{j-1} \times \text{count}_i \quad \text{for all } j = 1 \]
Theorem 1  Suppose $x = (\text{count}_1, \text{count}_2, \ldots, \text{count}_n)$ is the state primary state machines. Assume

$$f\text{Count}_j = \sum_i i^{j-1} \times \text{count}_i \text{ for all } j = 1..f$$

Given any $n$ values out of $y = (\text{count}_1, \text{count}_2, \ldots, \text{count}_n, f\text{Count}_1, f\text{Count}_2, \ldots, f\text{Count}_f)$ values in $x$ can be uniquely determined.

Proof Sketch:

- $y = xG$ where $G$ is $n \times (n + f)$ matrix $=[IV]$
  
  $V[i, j] = i^{j-1}, i = 1..N; j = 1..f$

- $y' = y$, suppressing the indices corresponding to the lost

- $M = \text{Delete corresponding columns in } G$

- $y' = xM$. 
• $M$ is a nonsingular matrix for all choices of the column of $G$.

• $x = y'M^{-1}$. 
Assume one Byzantine fault: need two fused copies Suppose changed by value $v$. Both $c$ and $v$ are unknown.

- $fcount_1$ differs from sum by $v$
- $fcount_2$ differs from $\sum_i count_i$ by $c \times v$.

$f/2$ errors can be located and corrected using $f$ fused copies.
Replication: $N$ primary state machines, $fN$ backup state machines

(1) Distinction between state machines and physical servers:

Can run $N$ backup state machines on one server.

Advantage of Fused Machines: Savings in storage. Disadvantage of Fused Machines: Recovery harder
Aggregation of Events
Using Order in Distributed Computing

\[ P(i) :: i = 1..n \]
\[ \text{int } \text{count}_i = 0; \]

On event entry\((v)\):
\[ \text{if } (v == i) \| (v == 0) \text{ count}_i = \text{count}_i + 1; \]

On event exit\((v)\):
\[ \text{if } (v == i) \| (v == 0) \text{ count}_i = \text{count}_i - 1; \]

\[ F(j) :: j = 1..f \]
\[ \text{int } \text{fCount}_j = 0; \]

On event entry\((i)\), for any \(i = 1..N\)
\[ \text{fCount}_j = \text{fCount}_j + i^{j-1}; \]

On event entry\((0)\)
\[ \text{fCount}_j = \text{fCount}_j + \sum_i i^{j-1}; \]

On event exit\((i)\) for any \(i = 1..N\)
\[ \text{fCount}_j = \text{fCount}_j - i^{j-1}; \]

On event exit\((0)\)
\[ \text{fCount}_j = \text{fCount}_j - \sum_i i^{j-1}; \]

Figure 3: Fusion of Counter State Machines
Fused Data Structures

Algorithms for Fusing arrays, linked lists, queues, hash tables, and updates with coding theory and efficient updates of backup data structures

- Use partial replication with coding theory
- Ensure efficient updates of backup data structures

ECE Dept., Univ. Texas at Austin
// Fused queue at $F(j)$

$fQueue$: array[0..M - 1] of int initially 0;

head, tail, size: array[1..n] of int initially 0;

append($i, v$);

if (size[$i$] == $M$)
    throw Exception("Full Queue");

$fQueue[tail[i]] = fQueue[tail[i]] + i^j - 1 \times v$;

tail[$i$] = (tail[$i$] + 1) mod $M$;

size[$i$] = size[$i$] + 1;

Figure 4: Fused Queue Implementation
\( P(i) :: i = 1..n \)

On receiving acquire from client \( pid \)

\[
\text{if } (\text{user} == 0) \{ \text{send(OK) to client } pid; \\
\text{user} = pid; \\
\text{send(USER, } i, \text{ user) to } F(j)'s; \} \\
\text{else } \{ \text{append(waiting, pid);} \\
\text{send(ADD-WAITING, } i, \text{ pid) to } F(j)'s; \}
\]

On receiving release

\[
\text{if } (\text{waiting.isEmpty()}) \{ \text{olduser} = \text{user}; \\
\text{user} = 0; \\
\text{send(USER, } i, \text{ user} - \text{olduser) to } F(j)'s; \} \\
\text{else } \{ \text{olduser} = \text{user;} \\
\text{user} = \text{waiting.head();} \\
\text{send(OK) to waiting.head();} \\
\text{waiting.removeHead();} \\
\text{send(USER, } i, \text{ user} - \text{olduser) to } F(j)'s; \\
\text{send(DEL-WAITING, } i, \text{ user) to } F(j)'s; \}
\]

\( F(j) :: j = 1..f \)

\[
\text{fuser:int initially 0;} \\
\text{fwaiting:fused queue initially 0;}
\]

On receiving (USER, \( i, \text{ val} \))

\[
fuser = fuser + i^{j-1} * \text{val;}
\]

On receiving (ADD-WAITING, \( i, \text{ pid} \))

\[
fwaiting.append(i, \text{ pid});
\]
Ricart and Agrawala’s Algorithm
$P_{i::i} = 1..n$

**var**

$pending$: array[1..n] of \{0,1\} init 0;
$myts$: integer initially 0;
$numOkay$: integer initially 0;
$wantCS$: integer initially 0;
$inCS$: integer initially 0;

receive(”requestCS”) from client:

$wantsCS := 1$;
$myts := logical\_clock$;
send (”request”, $myts$) to all (and $F(1)$);

receive(”request”, $d$) from $P_{q}$:

$pending[q] = 1$;
if ($wantCS == 0)$||$(d < myts)$ then
send okay to process $P_{q}$ (and $F(1)$);
$pending[q] = 0$;

receive(”okay”):

$numOkay := numOkay + 1$;
if ($numOkay = n - 1$) then
send(”grantedCS”) to client, $F(1)$;
in$CS := 1$;

receive(”releaseCS”) from client:
send(”releasedCS”, $myts$) to $F(1)$;
$myts, numOkay, wantCS, inCS := 0, 0, 0, 0$;
for $q \in \{1..n\}$ do
if ($pending[q]$) {
send okay to the process $q$;
Theorem 2 Let there be \( n \) primary state machines, each with 2P structures. There exists an algorithm with additional \( n + 1 \) RSMs that can tolerate a single Byzantine fault and has the same RSM overhead during normal operation and additional overhead during recovery.

Proof Sketch:

- one replica \( Q(i) \) for every \( P(i) \)
- a single fused state machine \( F(1) \)
- Normal Operation: Output by \( P(i) \) and \( Q(i) \) identical
- Byzantine Fault Detection: \( P(i) \) and \( Q(i) \) differ for any
- Byzantine Fault Correction: Use liar detection
Liar Detection

- $O(m)$ time to determine $O(1)$ size data different in $P(i)$
- Use $F(1)$ to determine who is correct
- No need to decode $F(1)$: Simply encode using value from $F(1)$
- Kill the liar
Theorem 3 There exists an algorithm with $fn + f$ backup machines that can tolerate $f$ Byzantine faults and has the same as the RSM approach during normal operation and additional overhead during recovery.

- Algorithm: $f$ copies for each primary state machine and unfused machines.

- Normal Operation: all $f + 1$ unfused copies result in the

- Case 1: single mismatched primary state machine
  Use liar detection algorithm

- Case 2: multiple mismatched primary state machine
  Can show that the copy with largest number of votes is
Other Fusion Related Work in PDSLAB

- Automatic Generation of Fused Finite State Machines
  [Balasubramanian, Ogale and Garg, IPDPS 09]
  [Balasubramanian and Garg, in progress]

- Efficient Algorithms for Fusion of Data Structures
  [Garg, ICDCS 07]
  [Balasubramanian and Garg, in progress]
Future Work

- Implementation of Algorithms for a Practical Server
- Different Fusion Operators