Fault-Tolerant Services in Distributed Systems Using

Vijay K. Garg

email: 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

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

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

On receiving release
\[
\text{if (waiting.isEmpty())}
\text{user = 0;}
\text{else } \{ \text{user = waiting.head();}
\text{send(OK) to user;}
\text{waiting.removeHead(); } \}\]
Tolerating Faults: Using Replication

\( f \): 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
- Agreement on the order Ensure that servers agree on the 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, $f = 1$
\[ P(i) :: i = 1..n \\
\quad \text{int } count_i = 0; \]

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

\[ F(1) :: \\
\quad \text{int } fCount_1 = 0; \]

On event \( \text{entry}(i) \), for any \( i \)
\[
\quad fCount_1 = fCount_1 + 1; \\
\]
On event \( \text{exit}(i) \) for any \( i \)
\[
\quad fCount_1 = fCount_1 - 1; \\
\]

Figure 1: Fusion of Counter State Machines
• Multiple faults
• More complex data structures
• Overflows
• Byzantine faults
Multiple Faults

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

On event entry(i), for any \( i \)
\[ fCount_j = fCount_j + i^{j-1}; \]

On event exit(i) for any \( i \)
\[ fCount_j = fCount_j - i^{j-1}; \]

Figure 2: Fusion of Counter State Machines

\[ fCount_2 = \sum_i i \ast \text{count}_i \]
\[ f\text{Count}_j = \sum_i i^{j-1} \times \text{count}_i \quad \text{for all } j = 1 \]
Theorem 1  Suppose $x = (\text{count}_1, \text{count}_2, \ldots, \text{count}_n)$ is the state primary state machines. Assume

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

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 columns $G$
• $x = y' M^{-1}$.
Assume one Byzantine fault: need two fused copies Suppose $c$ 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

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

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

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

On event \textit{entry}(\textit{i}), for any \( i = 1..N \)
\[ \textit{fCount}_j = \textit{fCount}_j + i^{j-1}; \]
On event \textit{entry}(0)
\[ \textit{fCount}_j = \textit{fCount}_j + \sum_i i^{j-1}; \]
On event \textit{exit}(\textit{i}) for any \( i = 1..N \)
\[ \textit{fCount}_j = \textit{fCount}_j - i^{j-1}; \]
On event \textit{exit}(0)
\[ \textit{fCount}_j = \textit{fCount}_j - \sum_i i^{j-1}; \]

Figure 3: Fusion of Counter State Machines
Fused Data Structures

Algorithms for Fusing arrays, linked lists, queues, hash tables, and Ogale 07, Balasubramanian and Garg 10]

- Use partial replication with coding theory
- Ensure efficient updates of backup data structures
// Fused queue at $F(j)$

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

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

$append(i, v)$;

\[
\text{if (size}[i] == M) \\
\quad \text{throw Exception(”Full Queue”);} \\
\text{fQueue[tail}[i]] = fQueue[tail][i] + ij^{−1} * v; \\
\text{tail}[i] = (tail}[i] + 1)\%M; \\
\text{size}[i] = size[i] + 1;
\]

$deleteH$;

\[
\text{if (size}[i] == 0) \\
\quad \text{throw Exception(”Empty Queue”);} \\
\text{head}[i] = (head}[i] + 1)\%M; \\
\text{size}[i] = size[i] - 1;
\]

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

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

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

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

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

On receiving (USER, \( i, val \))
  \[
  fuser = fuser + i^{j-1} \times val; \]

On receiving (ADD-WAITING, \( i, pid \))
  \[
  fwaiting.append(i, pid); \]
Ricart and Agrawala’s Algorithm
\[ P_i::i = 1..n \]

**var**

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

receive(”requestCS”) from client:

\[ wantsCS := 1; \]
\[ myts := \text{logical\_clock}; \]
\[ \text{send (”request”, myts) to all (and } F(1)); \]

receive(”request”, d) from \( P_q \):

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

receive(”okay”):

\[ numOkay := numOkay + 1; \]
\[ \text{if } (numOkay = n - 1) \text{ then} \]
\[ \text{send (”grantedCS”) to client, } F(1); \]
\[ inCS := 1; \]

receive(”releaseCS”) from client:

\[ \text{send (”releasedCS”, myts) to } F(1); \]
\[ myts, numOkay, wantCS, inCS := 0, 0, 0, 0; \]

for \( q \in \{1..n\} \) do

\[ \text{if } (pending[q]) \{ \]
\[ \text{send okay to the process } q; \]
\[ \} \]
Byzantine Faults

**Theorem 2** Let there be \( n \) primary state machines, each with unique structures. There exists an algorithm with additional \( n + 1 \) that can tolerate a single Byzantine fault and has the same the RSM approach 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
- 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 \textit{mismatched} primary state machine
  Use liar detection algorithm

- Case 2: multiple \textit{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