Fault-Tolerant Services in Distributed Systems Usin

Vijay K. Garg

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

Modeling Services in Distributed Syste

- Server: a Deterministic State Machine: not necessarily
- 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: int initially 0;
waiting: queue of int initially null;
On receiving acquire from client *pid*
if ($user == 0$) {
send(OK) to client *pid*; $user = pid$;}
else append(*waiting*, *pid*);
On receiving release
if (*waiting*.isEmpty())
user = 0;
else { user = *waiting*.head();
send(OK) to *user*;
waiting.removeHead(); }

Tolerating Faults: Using Replicatior

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 stat
- Agreement on the order Ensure that servers agree on t events

Byzantine faults: Keep identical $2f + 1$ replicas of the serv

- Use Voting If response is different, choose the response 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, $f = 1$

$P(i) :: i = 1..n$
 int $count_i = 0;$

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

$F(1) ::$
 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

Issues

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

Multiple Faults

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

- $$fCount_2 = \sum_i i * count_i$$

- $$fCount_j = \sum_i i^{j-1} * count_i \quad for\ all\ j =$$

Recovery from Crash Faults

Theorem 1 *Suppose* $\mathbf{x} = (count_1, count_2, , count_n)$ *is the*
primary state machines. Assume

$$fCount_j = \sum_i i^{j-1} * count_i \ for\ all\ j = 1..f$$

Given any n *values out of* $\mathbf{y}$
$= (count_1, count_2, ..count_n, fCount_1, fCount_2, ..fCount_f)$
values in $\mathbf{x}$ *can be uniquely determined.*

Proof Sketch:

- $\mathbf{y} = \mathbf{xG}$ where G is $n \times (n + f)$ matrix $= [IV]$
  $V[i, j] = i^{j-1}, i = 1..N; j = 1..f$
- $\mathbf{y}' = \mathbf{y}$, suppressing the indices corresponding to the los
- $\mathbf{M}$ = Delete corresponding columns in $\mathbf{G}$
- $y' = xM$.

- M is a nonsingular matrix for all choices of the column
G)
- $x = y' M^{-1}$.

Tolerating Byzantine Faults

Assume one Byzantine fault: need two fused copies Suppos
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 $c * v$.

$f/2$ errors can be located and corrected using f fused copie

State Machines vs Servers

Replication: N primary state machines, fN backup state m

(1) Distinction between state machines and physical server
Can run N backup state machines on one server.

Advantage of Fused Machines: Savings in storage. Disadva
Machines: Recovery harder

Aggregation of Events

$P(i) :: i = 1..n$
  int $count_i = 0$;

  On event $entry(v)$:
    if $(v == i)||(v == 0)$ $count_i = count_i +$
  On event $exit(v)$:
    if $(v == i)||(v == 0)$ $count_i = count_i -$

$F(j) :: j = 1..f$
  int $fCount_j = 0$;

  On event $entry(i)$, for any $i = 1..N$
    $fCount_j = fCount_j + i^{j-1}$;
  On event $entry(0)$
    $fCount_j = fCount_j + \sum_i i^{j-1}$;
  On event $exit(i)$ for any $i = 1..N$
    $fCount_j = fCount_j - i^{j-1}$; On eve
$exit(0)$
    $fCount_j = fCount_j - \sum_i i^{j-1}$;

Figure 3: Fusion of Counter State Machines

Fused Data Structures

Algorithms for Fusing arrays, linked lists, queues, hash tab
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);
    if (size[i] == M)
        throw Exception("Full Queue");
    fQueue[tail[i]] = fQueue[tail[i]] + i^(j−1) ∗ v;
    tail[i] = (tail[i] + 1)%M;
    size[i] = size[i] + 1;
```

```
deleteH
    if (s
        t
    fQu
    head
    size[

isEmpt
    retu
```

Figure 4: Fused Queue Implementation

$P(i) :: i = 1..n$
On receiving acquire from client *pid*
    if ($user == 0$) { send(OK) to client *pid*;
        $user = pid$;
        send(USER, i, $user$) to $F(j)$'s;}
    else { append($waiting, pid$);
        send(ADD-WAITING, i, pid) to $F(j)$'s;}

On receiving release
    if (waiting.isEmpty()) { $olduser = user$;
        $user = 0$;
        send(USER, i, $user - olduser$) to $F(j)$'s
    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, $user$) to $F(j)$'
}

$F(j) :: j = 1..f$
    $fuser$:int initially 0;
    $fwaiting$:fused queue initially 0;

On receiving (USER, i, val)
    $fuser = fuser + i^{j-1} * val$;

On receiving (ADD-WAITING, i, pid)
    $fwaiting.append(i, pid)$;

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);
             inCS := 1;

 receive("releaseCS") from client:
       send("releasedCS", myts) to F(1);
       myts, numOkay, wantCS, inCS := 0, 0, 0, 0;
       for q ∈ {1..n} do
             if (pending[q]) {
                   send okay to the process q;
```

Byzantine Faults

Theorem 2 *Let there be* n *primary state machines, each w 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 fro
- Kill the liar

Byzantine Faults: $f > 1$

Theorem 3 *There exists an algorithm with $fn + f$ backup machines that can tolerate f Byzantine faults and has the s as the RSM approach during normal operation and addition overhead during recovery.*

- Algorithm: f copies for each primary state machine an fused machines.
- Normal Operation: all $f + 1$ *unfused* copies result in th
- 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 PDSLA

- 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 [Gar
  ICDCS 07]
  [Balasubramanian and Garg, in progress]

Future Work

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