Distributed Predicate Detection: Applying Possibly and Definitely to Distributed Shared Memory Systems

Natasha M. Jarymowycz

December 9, 2002
Presentation Layout

- Predicate detection in a message passing system
  - introduction to problem
  - formal definition of system
  - protocol for solution

- Distributed shared memory
  - brief high-level overview
  - definition of sequential consistency
  - application of possibly and definitely

- Relaxing the memory model
Message Passing Distributed System

- $n$ processes $P_1, P_2, \ldots, P_n$
- Point-to-point, reliable, asynchronous, FIFO communication
- Message passing the only means of communication
- No global clock
A Global Predicate $\Phi$

- A boolean valued function defined on the *global state*

- Global state: a combination of all local $<$variable, value$>$ pairs

- Want to determine if $\Phi$ is true during a given execution

- Problem: In the absence of a global clock, no process has enough information about the systems to determine truth of $\Phi$ in well-defined manner
Possibly and Definitely

possibly $\Phi$: There is an execution of the system consistent with its observed behavior such that $\Phi$ was true at a point in that execution.

definitely $\Phi$: For all executions of the system consistent with its observed behavior, $\Phi$ was true at some point.
Each Process $P_i$:

- Local state $s_i$ changes when an event occurs
- Three types of events: local, message send, message receive
- Execution is modeled as a history
  
  $$h_i = s_0e_1s_1e_2s_2,...$$
Globally

- State $S = \langle s_1, s_2, \ldots, s_n \rangle$

- History $H = \langle h_1, h_2, \ldots, h_n \rangle$

- A *linearization* of a history, $L(H)$ is a sequence of global states and local events
  
  "Happens before" ordering determines legal sequences
  - maintain ordering of events within a process
  - message sends occur before receives
  - transitive
• $L(H)$ represents a possible ordering of events for a given execution. It is consistent with the information processes have about each other’s execution.

\[
L = \{S_{0,0}\} \quad x=4 \quad \{S_{1,0}\} \quad send \quad \{S_{2,0}\} \quad y=3 \quad \{S_{2,1}\} \quad recv \quad ...
\]

\[
= \{S_{0,0}\} \quad x=4 \quad \{S_{1,0}\} \quad y=3 \quad \{S_{1,1}\} \quad send \quad \{S_{2,1}\} \quad recv \quad ...
\]

\[
= \{S_{0,0}\} \quad y=3 \quad \{S_{0,1}\} \quad x=4 \quad \{S_{1,1}\} \quad send \quad \{S_{2,1}\} \quad recv \quad ...
\]
Example

\[ h_1: \{x=0\} x=4 \{4\} \text{send} m_1 \{4\} \text{recv} m_2 \{4\} x=8 \{8\} \]

\[ h_2: \{y=0\} y=3 \{3\} \text{send} m_2 \{3\} y=6 \{6\} \text{recv} m_1 \{6\} \]

\[ S_{0,0}: \{x=0, y=0\} \]
$h_1$: $\{x=0\}$ $x=4$ $\{4\}$ send $m_1$ $\{4\}$ recv $m_2$ $\{4\}$ $x=8$ $\{8\}$

$h_2$: $\{y=0\}$ $y=3$ $\{3\}$ send $m_2$ $\{3\}$ $y=6$ $\{6\}$ recv $m_1$ $\{6\}$

All linearizations for an execution can be represented as a lattice
Protocol

- Execution is augmented with *weak vector clocks*
  - vector $V$ of logical clocks
  - $P_i$ increments $V_i$ when it executes an event that may change $\Phi$
  - $V$ added to messages when sent
  - Update $V$ on receive

- Whenever $V$ changes at process $P_i$, message sent to monitor process with $V$ and state $s_i$

- Monitor builds state lattice using $V$ and $s_i$’s and performs state search, testing for satisfaction of $\Phi$ at each global state
  - *possibly*: true if any state satisfies $\Phi
– definitely: for each state, if Φ is not true then construct successors; if there are no more states to construct and last state is not the final state, then definitely is true

```
(0,0)
  
(1,0)  (0,1)
  
(2,0)  (1,1)  (0,2)
  
(2,1)  (1,2)  (0,3)
  
(2,2)  (1,3)
  
(3,2)  (2,3)
  
(4,2)  (3,3)  (2,4)
  
...   (4,3)  (3,4)
  
...   (4,4)  ...
  
...   ...
```
Distributed Shared Memory

- Processes communicate via shared memory
- No messages
- Memory consistency model determines when concurrent reads and writes are seen by other processes
  - 2 processes can try to write to the same variable concurrently
  - Writes occur over an interval of time
  - Read returns the value from the last write
Sequential Consistency

- Extension of multi-process shared memory

- Formal definition:
  There is an interleaving of all reads and writes so that a read returns
  the last value written; reads and writes of each process occur in
  program order

  Process 1:                      Process 2:
  write(x, 1)                     write(y, 1)
  read(y)  0                      read(x)  ?

  ordering: write(x, 1)  read(y) 0  write(y, 1)  read(x) 1
Predicate Detection

- Global predicate: defined on *global* memory values

- Local history with local state information is insufficient for determining global memory values

\[
\begin{align*}
\text{Process 1:} & \quad \text{Process 2:} & \quad \text{Process 3:} \\
& \quad x=4; \ y=6; & \quad \text{write}(x, 6) \\
& \quad \text{write}(y, 2) & \quad \text{read}(y) 6 \\
& \quad \text{read}(x) 6 & \quad \text{read}(x) 6 \\
\Phi = (x+y > 10) & & \\
\end{align*}
\]

- Linearization respects ordering: value cannot be read before it’s written and read returns last value written in sequence
By definition, sequential consistency only allows executions for which a linearization exists (interleaving); may be more than one.

\[ L(H): \text{read}(y) \text{6 write}(y,2) \text{ write}(x,6) \text{ read}(x,6) \]
\[ \text{read}(y) \text{6 write}(x,6) \text{ write}(y,2) \text{ read}(x,6) \]

possibly but not definitely

Underlying DSM system aids in addition of timestamp info which may limit possible linearizations.
Protocol

- Like message passing system
  - weak vector timestamps
  - processes send timestamp and event to monitor
  - monitor builds state lattice as before

- Monitor must reconstruct global memory values
  - store writes to each global variable (of interest)
  - at each state there is only one last written value
  - last write used to determine value for predicate detection
• Lattice will allow some linearizations which don’t respect the ordering as a consequence of the implementation of timestamps
  – faulty ordering allowed by sequential consistency as long as reading process has not already read new value
  – essentially, sequential consistency allows a certain level of indeterminacy which is evident in timestamp
  – to correct: use last written value for determining global memory values
**Example**

**Process 1:**
\[(1,0,0)\text{write}(x, 6)\]

**Process 2:**
\[x=4; \ y=6;\]
\[(0,1,0)\text{write}(y, 2)\]
\[\Phi = (x+y > 10)\]

**Process 3:**
\[(0,0,1)\text{read}(y) \ 6\]
\[(1,0,2)\text{read}(x) \ 6\]
Relaxed Memory Consistency

- Sequential consistency may be overly restrictive, costly to enforce

- Allow some processes to see different orderings of some writes
  For example:
  - totally order all writes by one process and writes to a given variable
    This allows writes to different variables by different processes to be seen out of order
  - order writes based on causality
    Concurrent writes can be seen in either order by processes
  - hybrid models: distinguish different types of accesses
    Synchronization accesses are sequentially consistent and impose ordering rules on ordinary reads and writes
• Individual processes always respect own program ordering

• Consequence of relaxing memory: there can be more than one legitimate value to be returned by a read

Process 1:  
write(x, 1)  
read(y) 0

Process 2:  
write(y, 1)  
read(x) ?

• Linearization not possible:  
In the above example, one of the writes must come first  
Essentially, each process has own linearization of global history

• Solution: modify definitions of possibly and definitely to account for set of values for each variable
Definitions

- A global state in the lattice satisfies possibly if *any* of the legitimate possible values of the variables satisfy $\Phi$

- If a global state satisfies possibly, then possibly is true

- A global state in the lattice satisfies definitely if all of the possible legitimate possible values of the variable satisfy $\Phi$.

- For definitely to be true, all paths through the lattice must pass through states which satisfy definitely.
Example

Process 1:
(1,0) write(x, 3)
(2,0) write(y, 4)

Process 2:
(0,1) write(x, 2)
(1,2) read(x) 3

Φ : x = 3
Analysis

- With relaxed consistency models, definitely is harder to satisfy: values tend to linger in system

- Algorithm at monitor can be modified based upon memory model: skip construction of states till "bad" value is overwritten

- Example uses weak memory:
  - accesses to synchronization variables are sequentially consistent
  - before a read or write to a synchronization variable, all previous accesses are propagated
  - can not access a global variable until access to previous synchronization variable has completed
Process 1:

(1,0,0) write(x, 1)
(2,0,0) write(y, 2)
(3,0,0) write(sync, 4)

Process 2:

(0,1,0) write(x, 2)
(3,2,0) write(y, 3)
(3,3,1) write(sync, 6)

Process 3:

(3,0,1) read(sync) 4
(3,0,2) read(y) 2
Assume $x=1$ implies $\Phi$ is false

In order to remove 1 from possible values for $x$, write must be later than $\text{write}(x, 1)$ (non-concurrent)

Later write must have a timestamp greater than $(1,0,0)$

The $\text{write}(x, 1)$ must have completed at all processes before later write

In the example, monitor can avoid constructing states until execution of another write to $x$ - if write satisfies conditions for non-concurrency, then start construction at that point in lattice
Summary

- Definition of possibly and definitely
- Application to distributed shared memory - sequential consistency
- Modification for relaxed consistency
- Other solutions for relaxed consistency?