Reliable Speculative Processing of Out-of-Order Event Streams in Generic Publish/Subscribe Middlewares

Christopher Mutschler, Michael Philippsen

The 7th ACM International Conference on Distributed Event-Based Systems (DEBS' 13)
July 2nd, 2013. Arlington, Texas
Contents

- Introduction
  - Problem description
  - Motivation
  - System environment, time model semantics, event ordering basics
- Our contribution: Speculative Event Processing
  - Event emission and replay
  - State recovery
  - Event retraction
  - Runtime adaptation
- Experimental Results
- Conclusion
Introduction

Problem description

Adapted idea from "Logical View on EPTS Reference Architecture" (Paschke et al. 2012)
Introduction
Problem description (cont’d)

- Events are generated at different points in the network and (different) delays become inevitable and the event ordering is incorrect.
- Deal with out-of-order events within the event detectors, or…
- Take one of two possible resorts for middleware implementation:
  - Buffering middlewares: conservative → save.
    - Withhold events to guarantee correctly emitted events to the detectors.
    - Causes high/unnecessary latencies; additional CPU power cannot reduce latency.
  - Speculative middlewares: no buffering → fast.
    - Process events speculatively and retract them on out-of-order events.
    - Unbounded retraction cascades lead to CPU overload and event processing may eventually fail (because event load cannot be satisfied).
Introduction
Motivation: combining the advantages of both worlds

- Idea: add a speculative processing technique to a buffering middleware.
  - Sort most of the (out-of-order) events.
  - Prematurely process events that will be emitted soon.
  - When the stream must be replayed, restore the event detector’s state.
  - Adapt the degree of speculation at runtime to fit the available resources.

- The most generic solution must adhere to the following restrictions:
  - The middleware can neither exploit the events’ semantics nor their use by detectors as this is highly application specific.
  - The middleware may not access the detector’s internal implementation.
  - Event detection must always be reliable, i.e., false-positive or false-negative events must be avoided to prevent system failures (e.g. because of invalid states).
Introduction
Time model semantics and basic assumptions

Event type, instance, and time stamps:

- $\text{ID}$ the event is an instance of event type $\text{ID}$,
- $ts$ the occurrence time stamp (same discrete time source),
- $dts$ the detection time stamp,
- $ats$ the arrival time stamp.

Node’s time synchronization:

- $clk$ node’s internal clock, either assigned by wall-clock time retrieval or by events of type $\text{ID}_{\text{clk}}$. 
Introduction

Event ordering basics: dynamic K-slack

- We buffer the detector’s events, and whenever we update $clk$:
  1. Calculate delay of any event $\delta(e_i) = clk - e_i.ts$ received since last update.
  2. Update $K = \max_i [\delta(e_i)]$ (if necessary).
  3. Emit and purge any event that satisfies $e_i.ts + K \leq clk$. 

![Diagram showing event ordering and delay calculation](image)
Speculative Event Processing
Terminology

- **Event emission and replay:**
  - Ordering units *emit* events to the event detector for processing.
  - Events are emitted *prematurely* if they are emitted although K-slack not holds.
  - Event detectors *generate* events and send them to the middleware.
  - Whenever ordering units detect miss-speculation events are *replayed*.

- **Event retraction:**
  - Whenever ordering units detect miss-speculation generated events must be *retracted* from upper level event detectors’ ordering units.
  - Returns in cascades throughout the detector hierarchy.
Speculative Event Processing

Event emission and replay

- Emit any event $e_i$ that satisfies

  $$ e_i . ts + \alpha \cdot K \leq clk, \quad 0 \leq \alpha \leq 1. $$

- Events are no longer emitted and purged at the same time.
- Events are prematurely emitted according to the attenuation factor $\alpha$.
- Purge of events only when K-slack holds.
- $\alpha = 1$ is equivalent to (full) buffering
- $\alpha = 0$ is equivalent to full speculation.
Speculative Event Processing
Event emission and replay (cont’d)

- The buffer is enhanced with an *already emitted pointer* that references to the last event in the buffer that has already been prematurely emitted.

- Newly arriving events are inserted into the sorting buffer.
  - Case 1: time stamp of new event is larger than that of the AEP → ok.
  - Case 2: time stamp of new event is smaller than that of the AEP → bad.

- For case 2 we reset the AEP and replay the event stream.
Speculative Event Processing

Event emission and replay (cont’d)

\[ \alpha = \frac{1}{3} \]

\(\text{unsorted event stream / input}\)

\(\text{sorted event stream / output}\)

\(\text{id}\)

\(\begin{array}{cccccccccccc}
0 & 2 & 1 & 3 & 4 & 5 & 8 & 7 & 11 & 10 & 12 & 9 \\
\end{array}\)

\(\text{ts}\)

\(\begin{array}{cccccccccccc}
0 & 2 & 1 & 3 & 4 & 5 & 8 & 7 & 11 & 10 & 12 & 9 \\
\end{array}\)

\(\begin{array}{cccccccccccc}
A0 & A0 & C1 & C1 & A2 & A2 & C5 & C5 & C5 & C5 & A6 & A6 \\
C1 & C1 & B4 & B4 & B4 & B4 & B8 & B8 & B8 & B8 & B8 & C9 \\
\end{array}\)

\(K/\text{clk}_4\)

\(\begin{array}{cccccccccccc}
0 & 2 & 0 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}\)

head

\(\text{tail}\)

\(\begin{array}{cccccccccccc}
0 & 2 & 0 & 2 & 2 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\
\end{array}\)

\(\begin{array}{cccccccccccc}
A12 & A12 & A12 & A12 & A12 & A12 & A12 & A12 & A12 & A12 & A12 & A12 \\
\end{array}\)
Speculative Event Processing

State recovery

- When events are replayed the internal state of the event detector is affected.
  1. Event detectors hold a list of states and process *retraction events*.
  2. Middleware asks for / sets the state of the event detector.

- The 2\textsuperscript{nd} approach has several advantages:
  - Event detectors normally have no clue that ordering units exist.
  - The detector has no clue about $K$ and $\alpha$, and hence how many states to keep.
  - Retraction cascades can be interrupted earlier (triggered by the middleware).
  - Side-effects are reduced.

- Solution:
  - Middleware handles state backup and recovery!
Speculative Event Processing
State recovery (cont’d)

- Snapshots are inserted as events with time stamps of upcoming events.
- Represent the event detector state between two prematurely emitted events.
- Whenever events are replayed and AEP is reset, the snapshot event is provided to the event detector for state recovery (only the first is taken).

Requirements:
1. Detector must provide snapshot on demand.
2. Detector must restore to snapshot on demand.

- Snapshot events are purged as any other events.
Speculative Event Processing
State recovery (cont’d)

unsorted event stream / input

\[ \alpha = \frac{1}{3} \]

sorted event stream / output
Speculative Event Processing

Event retraction

- Event detectors get restored correctly if speculation was too hasty.
- But detectors may have generated events based on the wrong order.
- We must retract mistakenly generated events from upper level ordering units.

Two ways for hierarchy retraction: full retraction and on-demand retraction.
Speculative Event Processing

Event retraction: full retraction

- Key idea: retract all events generated as a result of premature emissions.
- Fastest possible retraction technique (essentially state-of-the-art).
- We need to store a list of *generated* events based on premature events and send a (conceptual) retraction event for each of them.
- Upper level ordering units relocate their AEP and perform retraction/replay.
- If states change often on input events this is the fastest retraction technique.
Speculative Event Processing

Event retraction: on-demand retraction

- If states rarely change most of the retraction work is useless since exactly the same detection work is performed again (e.g. state-less event detectors).
- The achievable degree of speculation and the efficiency of full retraction strongly depends on the number of retraction:
  - Snapshots are equal: the same events will be generated.
  - Generated events are equal: no retraction need on upper level detectors.
- Works more efficient for state-less (or less-state change) event detectors.
Speculative Event Processing

Runtime $\alpha$-adaptation

- $\alpha$ controls the degree of speculation, i.e., the detection latency of events.
- Tune $\alpha$ to exploit CPU power but do not exaggerate and overload the system.
- The idea:
  - Start with an initial $\alpha = 1$, i.e., conservative buffering.
  - Use CPU runtime measurements and decrease $\alpha$.
  - Increase $\alpha$ quickly, i.e., set $\alpha = 1$, if the system consumes too much CPU.
Speculative Event Processing
Runtime \(\alpha\)-adaptation (cont’d)

- Measurement interval of length \(t_{\text{span}}\).
- Sum of worker thread busy times \(t_{\text{busy}}\).
- On expiration of \(t_{\text{span}}\) we derive the busy factor \(b_c = \frac{t_{\text{busy}}}{t_{\text{span}}}\).
- The aim is that \(b_c\) reaches a value in \([b_l; b_u]\), the target zone.
- Use the congestion control mechanism known from TCP:
  - Halve \(\alpha\) as long as \(b_c\) is lower than \(b_l\) (double window size when no packets lost).
  - Increase \(\alpha = 1\) on critical CPU load and save the lowest/last value \(\alpha_l\).
  - If \(\alpha\) should be set lower than the bisection line \((1-\alpha_l)/2\) it is only decreased by \(\alpha_s\) for slow convergence to target zone (congestion avoidance).
Experimental Results

Test setup

- Sensor data from a Realtime Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany.
- Evaluation data at a rate of approx. 2.67 Mbytes/sec.
- Platform of Intel Xeon E5560 Quad Core CPUs @ 2.80 GHz, 64 GB RAM
- Middleware implementation ~ 15k LOC, detectors ~ 16k LOC.
- 70 event detectors, 750 different event types.
- To focus on the results we only use a single machine.
- We chose a representative snippet of 65 sec. to delve into the results.
Experimental Results
Latency reduction

- Latency measurements of a *pass*-detector, subscribes 6 event types.
- Dynamic K-slack updates K upon ordering mistakes, ends up at 1458ms; average latency for the 65 sec. was 1276ms.
- Spec. processing with $\alpha$-adaptation has an avg. latency of ~ 800ms.
- CPU load peaks result in resets of $\alpha$.
- 40% less than dynamic K-slack.
Experimental Results

Runtime $\alpha$-adaptation

- Zoom into the first 20 sec. of the stream replay.
- Busy factor target zone of $[0.8;0.9]$, $t_{span} = 0.5$ sec.
- Reaching $\alpha = 0.125$ and stabilizing in target interval after 7 seconds.
- After 8.5 sec. $b_c = 0.95$ (91% CPU) and $\alpha$ is instantly set to 1.
- After 16 sec. halving stops at the bisection line at $(1.0-0.125)/2=0.43$ and takes small steps of $\alpha_s = 0.05$.
- $b_c$ slightly overestimates (task switches, thread sleeping, …)
Experimental Results

Resource consumption

- Replay 65 sec. snippet three times (pure, full, dynamic), only 1 worker thread.
- Pure buffering exhibits lazy CPU load between 50-70% all the time.
- Full speculation has CPU loads of 100% several times (maybe even all the time) causing ripple effects in the measured event delays and the retraction cascades.
- Dynamic speculation achieves the good latencies from before by adaptively using available resources.

Memory consumption:

- Pure: 1,120 KB, full: 21,100 KB, dynamic: 14,850 KB.
- Most states are smaller than 50 bytes (one unique was 800 KB).
Conclusion

Conclusion and future work

- Speculative buffer extension achieves reliable and low-latency processing.
- Buffering middlewares may use the speculation to reduce detection latency.
- No a-priori configuration, no information on internal implementation needed.
- Speculation adaptively approaches the minimal latency by using resources.
- The evaluation proves that the speculation works in busy and lazy situations.
- Reduction of detection latency by 40%.

Future Work:

- Refine $\alpha$-adaptation to incorporate individual event loads and detectors.
- Investigation on choosing the appropriate retraction technique at runtime.