On the design of multi-tagged event queues and their effects on Distributed services and Computing*

February 25, 2000

Contents

1 Introduction

Events are an indication of an interesting occurence. Events point to nuggets of information which are related to the event itself, and help us understand the event completely. When we refer to an event we refer to the occurence and the information it points to. The information contained in the event comprises of

- Event type i.e. the occurence.
- Attribute information.
- Control information.

The attribute information comprises of tags which specify the attributes associated with the event type while the control information specifies the constraints associated with that event viz. ordering, stability. Thus say a person needs to sell stock A - the selling is the event, the general information is his account profile while the control information could be a indication that he wants guaranteed delivery of the event.

Events trigger *actions*, which in turn can trigger events. The event and the associated actions taken by any part of the system share the *cause-effect* relationship. Actions are taken based on the event type and the information contained in the event. The action taken at any node could be influenced not only by different causes but by subsequent effects too.

This paper, is about events, the organization, retrieval and specification of attributes and constraints associated with that event. This paper is also about event queues comprising of the aforementioned events. We attempt to provide a motivation and a solution for how the queues can exist

*Northeast Parallel Architectures Center, Syracuse University, Syracuse NY 13210

over the network. Issues pertaining to replication and consistency will also be discussed. Finally this paper is also about the design of an event service, specifying client and server rules, which would use these events and the event queues.

2 Specifications

We now try to specify our problem. In section 2.1 we present our model of the system in which we intend to solve the problem. In section 2.2 we formally specify our problem. Sections 2.3 and 2.4 deal with the assumptions that we make in our formalism's and the properties that the system and it components must conform to during execution.

2.1 System Model

The system comprises of a finite (possibly unbounded) set of *server nodes*, which are strongly connected (via some inter-connection network). Special nodes called *client nodes*, can be attached to any of the server nodes in the network. Let $\mathbf{C}$ denote the set of client nodes present in the system. The nodes, servers and clients, communicate by sending events through the network. This communication is asynchronous i.e. there is no bound on communication delays. Also the events can be lost or delayed. A server node execution comprises of a sequence of actions, each action corresponding to the execution of a step as defined by the automaton associated with the server node. We denote the action of a node sending an event e as $send(e)$. At the receiving end the action of consuming an event e is $deliver(e)$. Server nodes relay the events to the client nodes, we denote this action $relay(e)$.

The failures we are presently looking into are node failures (client and server nodes) and link failures. The server node failures have crash-failure semantics and could be one of the following:

(a) Crash - A faulty node stops prematurely and does nothing from that point on.

(b) Send Omission - A faulty node stops prematurely, or intermittently omits to send messages it was supposed to send, or both.

(c) Receive Omission - A faulty node stops prematurely, or intermittently omits messages sent to it, or both.

(d) General Omission - A faulty node is subject to send and receive omission failures.

Link Failures are of two types:

(a) Crash - A faulty link stops transporting messages. Before stopping however it behaves correctly.

(b) Omission - A faulty link intermittently omits transporting messages sent through it.

2.2 The event service problem

Client nodes can issue and deliver events. Any arbitrary event e contains implicit or explict information regarding the client nodes which should deliver the event. We denote by $L_e \subseteq \mathbf{C}$ this destination list of client nodes associated with an event e. The dessimination of events can be one-to-one or one-to-many. Also, the client nodes can have intermittent connection semantics and can roam around the network, attaching itself to possibly different server nodes during a specific execution trace of the system.

For an execution σ of the system, we denote by E_σ the set of all events that were issued by the client nodes. Let $E^i_\sigma \subseteq E_\sigma$ be the set of events e^i_σ that should be relayed by the network and delivered by client node c_i in the execution σ. During an execution trace σ client node c_i can *join* and *leave* the system. Node c_i could *recover* from *failures* which were listed in Section 2.1. Besides this, as mentioned earlier client nodes can roam (a combination of leave from an existing location and join at another location) over the network. A combination of join-leave, join-crash, recover-leave and recover-crash constitutes an *incarnation* of c_i within execution trace σ. We refer to these different incarnations, $x \in X = 1, 2, 3...$, of c_i in execution trace σ as $c_i(x, \sigma)$.

The problem pertains to ensuring the delivery of all the events in E^i_σ during σ irrespective of node failures and location transcience of the client node c_i across $c_i(x, \sigma)$. In more formal terms if node c_i has n incarnations in execution σ then

$$\sum_{x=1}^{n} c_i(x, \sigma).deliveredEvents = E^i_\sigma.$$

All delivered events $e^i_\sigma \in E^i_\sigma$ must ofcourse satisfy the causal constraints that exist between them prior to delivery.

2.3 Assumptions

(a) Every event e is unique.

(b) The links connecting the nodes do not create events.

2.4 Properties

(a) A client node can deliver e, only if e was previously issued.

(b) A client node delivers an event e only if that event satisfies the constraints specified in its control information.

(c) If an event e is to be delivered by client nodes $c, c' \in L_e$, then if c delivers e then c' will deliver event e.

(d) For two events e and e' issued by the same client node c, if a client node delivers e before e', then no client node delivers e' before e.

(e) For two events e and e' issued by nodes c and c' respectively, if a node delivers e before e', then no node delivers e' before e.

Properties (d) and (e) pertain to the causal precedence relation $\rightarrow$ between two events e, e', and can be stated as follows $\forall c_i \in L_e \bigcap L_{e'}$ if $e \rightarrow e'$ then $e.deliver() \rightarrow e'.deliver()$. $\rightarrow$ is transitive i.e. if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e' \rightarrow e''$.

3 Unique Events

Associated with every event e sent by client nodes in the system is an event-ID, denoted $e.id$, which uniquely determines the event e, from any other event e' in the system. These ID's thus have the requirement that they be unique in both space and time. Clients in the system are assigned Ids, ClientID, based on the type of information issued and other factors such as location, application domain etc. To sum it up client's use pre-assigned Ids while sending events. This reduces the uniqueness problem, alluded earlier to a point in space. The discussion further down implies that the problem has been reduced to this point in space.

Associating a timestamp, $e.timeStamp$, with every event e issued restricts the rate (for uniquely identifiable events) of events sent by the client to one event per granularity of the clock of the underlying system. Resorting to sending events without a timestamp, but with increasing sequence numbers, $e.sequenceNumber$, being assigned to every sent event results in the ability to send events at a rate independent of the underlying clock. However, such an approach results in the following drawbacks

a) If the client node issues an infinite number of events, and also since the sequence numbers are monotonically increasing, the sequence number assigned to events could get arbitrarily large i.e. $e.sequenceNumber \rightarrow \infty$.

b) Also, if the client node were to recover from a crash failure it would need to issue events starting from the sequence number of the last event prior to the failure, since otherwise event would be deemed a duplicate otherwise.

A combination of timestamp and sequence numbers solves these problems. The timestamp is calculated the first time a client node starts up, and is also calculated after sending a certain number of events $sequenceNumber.MAX$. In this case the maximum sending rate is related to both $sequenceNumber.MAX$ and the granularity of the clock of the underlying system. Thus the event ID comprises of a tuple of the following named data fields : $e.PubID$, $e.timeStamp$ and $e.sequenceNumber$. Events issued with different times t1 and t2 indicate which event was issued earlier, for events with the same timestamp the greater the timestamp the later the event was issued.

References

[BBT96] Anindya Basu, Bernadette Charron Bost, and Sam Toueg. Solving problems in the presence of process crashes and lossy links. Technical Report TR 96-1609, Dept. Of Computer Science, Cornell University, Ithaca, NY-14853, September 1996.

[BCM+99] Gurudutt Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Rob Strom, and Daniel Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[BF96] Ken Birman and Roy Friedman. Trading consistency for availability in distributed systems. Technical Report TR96-1579, Dept. Of Computer Science, Cornell University, Ithaca, NY-14853, April 1996.

[Bir85] Kenneth Birman. Replication and fault tolerance in the isis system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, Orcas Island, WA USA, 1985.

[Bir93] Kenneth Birman. A response to cheriton and skeen's criticism of causal and totally ordered communication. Technical Report TR 93-1390, Dept. Of Computer Science, Cornell University, Ithaca, NY 14853, October 1993.

[BM89] Kenneth Birman and Keith Marzullo. The role of order in distributed programs. Technical Report TR 89-1001, Dept. Of Computer Science, Cornell University, Ithaca, NY 14853, May 1989.

[GRVB97] Katherine Guo, Robbert Renesse, Werner Vogels, and Ken Birman. Hierarchical message stability tracking protocols. Technical Report TR97-1647, Dept. Of Computer Science, Cornell University, Ithaca, NY 14853, 1997.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Dept. Of Computer Science, Cornell University, Ithaca, NY-14853, May 1994.