Snapshot Processing in Streaming Environments

Daniel M. Zimmerman and K. Mani Chandy

Department of Computer Science, California Institute of Technology
Mail Stop 256-80, Pasadena, California 91125, U.S.A.
{dmz, mani}@cs.caltech.edu

I. INTRODUCTION

Computational issues related to streaming data, and in particular the monitoring and rapid correlation of multiple sources of streaming data, are becoming increasingly important in contexts ranging from business processes to crisis detection. Applications include automated commodities trading (streams of stock and commodity ticker data), medical monitoring (streams of medical information from instruments worn by or in the vicinity of patients), and the detection of security threats such as biological and chemical weapons (streams of readings from radiation and biohazard detectors, intelligence services, immigration checkpoints, and more). For example, a government system to detect bioterror attacks must correlate multiple streams of possibly low-confidence data from sensors and local and national public health information networks with cues from indicators such as news and government sources indicating geographical locations, tactics and timing of possible attacks. The results of this correlation trigger appropriate responses, such as flagging information for more in-depth analysis or sending alerts to public health officials.

Monitoring and correlation applications of this type are ideal for deployment on distributed computing grids, because they have high transaction throughput, require low latency, and can be partitioned into sets of small communicating computations with regular communication patterns. An important consideration in these applications is the need to ensure that, at any given time, computations are carried out on an accurate— or at least close to accurate—picture of the environment being monitored. One way of doing this, which we call *snapshot processing*, is to treat collections of events that occur at approximately the same time as representing a *global snapshot*—a valid state—of the environment. Computation on the resulting series of snapshots is much like computation on a real-time video of the entire environment. We briefly describe our model for these stream processing computations and introduce the concept of snapshot processing.

II. GRAPH MODEL OF STREAM PROCESSING

In a given stream processing system, there may be many data sources. The data can be unstructured (as in news stories or images), structured (as from databases or Web Services, where data schemas are well defined), or partially structured. The computation can be represented by a directed acyclic graph in which the nodes represent computational elements and the directed edges represent data flowing between elements. For example, a node may represent the evaluation of natural language text messages to determine their relevance to a particular situation. A node may also represent an incremental statistical analysis such as a regression or a change-detection algorithm, or even a simple Boolean operation such as conjunction. Some nodes may execute multiple models to determine which one best explains the current data. Thus, the granularity of nodes varies widely. The *sources* (nodes without predecessors) of the graph represent data sources, such as cameras and sensors, and the *sinks* (nodes without successors) represent responses, such as email alerts and emergency shutdowns.

Information from multiple streams is correlated as data arrives. The state of any given sensor or information source changes relatively slowly; at each instant in time the global state may be exactly the same as, or differ only slightly from, the state at the previous instant. Therefore, efficient algorithms communicate and compute incrementally based on the changes between consecutive states.

Data may be noisy. There may be a delay between the instant at which a measurement is made by a sensor and the instant at which a message containing that measurement data arrives at the correlation engine. Therefore errors—*false positives* where a response is generated when none should be, and *false negatives* where no response is generated, or a response is generated later than it should be—are possible. The challenge is to design systems that have tolerable error rates, process streams of data from all the data sources at the rates at which they are generated, and have adequate response times.

III. SNAPSHOT PROCESSING

Assume that the stream processing system is a distributed system—either deployed on a grid or "emulated" within a single shared-memory machine—having exactly the same structure as the computation graph: one processor (or process) for each node, and a message channel between nodes for each directed edge. There are other possible stream processing system architectures, but we focus on this one at present.

A typical method of processing streams is *on-arrival processing*: each process remains idle until one of its input channels becomes non-empty, at which point it executes a step by removing a message from one of its input channels and carrying out the incremental computation associated with that message. This execution may result in the generation of messages to other processes.

Snapshot processing, by contrast, attempts to perform computations on one consistent global state at a time. Each message generated by a data source is assigned a timestamp (we assume lightweight clock synchronization among the processes, and therefore bounded clock drift). Time is partitioned into a sequence of contiguous intervals of length D; all messages with timestamps in the interval $[k \times D, (k+1) \times D)$ (called *computation phase*, or simply *phase*, k), are treated as though they reflect the kth global state, and are processed as an atomic unit. If multiple messages on the same channel have timestamps in phase k, all but the last are discarded.

The choice of D depends on the rates of change of sensor values. A D that is too large may encompass several changes in the measurements made by one sensor; since only the last of these measurements will be used to represent the entire interval, this can result in less accurate estimates of global states. A D that is too small may result in large numbers of process activations and a consequent performance penalty.

Consider two extreme cases of computation graphs to gain intuition into the relative performances of on-arrival and snapshot processing algorithms. The first consists of a set of nodes with at most one input edge (no fan-in); it is a forest of independent subgraphs (trees), where the root of each tree is a source node and the leaves of the tree are sink nodes. Since different trees do not share nodes, this represents a computation in which each stream is analyzed independently. The second consists of a set of S source nodes followed by K *layers*, where each layer has M nodes and each node in a layer has incoming edges from all nodes in the previous layer (significant fan-in). The M nodes in each layer and M^2 edges between each pair of layers give the graph a total of $(K-1) \times M^2 + S \times M$ edges. This represents a computation in which there is substantial correlation across streams.

Each message arriving at each node may result in that node sending a message on each of its outgoing edges. Consider the steps taken when processing a message at a source node of each of the two graphs. In the first graph, a message arriving at a source node (the root of a tree) may cause messages to be propagated along its outgoing edges; thus the number of node activations is the number of nodes in the tree, for both on-arrival and snapshot processing. In the second graph, however, a message arriving at a source node is sent to M nodes in the first layer. With on-arrival processing, each of these M messages can cause messages to be sent to all M nodes in the second layer, for a total of M^2 additional messages, and the progression continues. Thus, a total of $\Sigma_{j=0}^{j=K} M^j$ $((M^{K+1}-1)/(M-1))$ messages could be generated by each message arriving from a data source, and each could cause a node activation; for a small graph with $K = M = 5$, this is 3,906 messages and 3,906 node activations. With snapshot processing, on the other hand, each message channel is used at most once per phase and each node is activated at most once per phase. Thus the number of node activations is at most $K \times M$, and the generated number of messages is at most $(K-1) \times M^2$; for $K = M = 5$, this is 100 messages and 25 node activations. Clearly, for this type of graph, snapshot processing requires far less computation and communication.

Errors in the system can arise in two ways: (1) they can be caused by the signaling and data fusion algorithms; and (2) they can be caused by the underlying architecture (sensor errors, timestamp drift, message delays between sensors and computation elements). Neither on-arrival nor snapshot processing deals with the second type of error, but the algorithms differ in the way they deal with the first. With on-arrival processing, a node may carry out a computation on a message with a later timestamp before it carries out a computation on a message with an earlier timestamp, because messages are propagated along the edges of the graph as they are computed and some paths may execute faster than others. This may cause the computation to use out-of-date or incorrect information. Snapshot processing eliminates this out-of-order execution problem by forcing the computations to remain in phase; however, errors may be introduced by treating all measurements within a phase as having occurred simultaneously. The frequency and severity of these errors are dependent on the choice of phase interval D.

The performance benefits of snapshot processing and the fact that it is not susceptible to out-of-order execution errors are good reasons to prefer it over on-arrival processing for streaming data correlation. However, the in-order execution and noninterfering computations of snapshots do not come for free; like the set of ACID properties for database transactions, snapshot processing requires specialized scheduling algorithms. Note that "scheduling" in this context does not refer to the assignment of tasks to resources in a distributed computing grid, also known as resource allocation, which has been studied extensively; instead, it refers to the scheduling of individual operations within a larger computation to enable concurrency while preserving data dependencies. We have developed multiple scheduling algorithms for snapshot processing. These include *null-message scheduling*, which uses ideas from distributed simulation [1]; *m-threshold scheduling*, which is described in a previous paper [2]; and *layer scheduling*, which will be described in a forthcoming paper. The latter two are based on the idea of a Δ-*dataflow* [3] graph, in which nodes send messages to their successors in the graph only when their computations generate information that does not conform to models shared with those successors.

REFERENCES

[1] J. Misra, "Distributed discrete event simulation," *ACM Computing Surveys*, vol. 18, no. 1, pp. 39–65, Mar. 1986.
[2] D. M. Zimmerman and K. M. Chandy, "A parallel algorithm for correlating event streams," in *19th International Parallel & Distributed Programming Symposium (IPDPS 2005)*, Apr. 2005.
[3] R. Manohar and K. M. Chandy, "Δ-Dataflow networks for event stream processing," in *IASTED International Conference on Parallel and Distributed Computing and Systems*, Nov. 2004.