Scalable Distributed Stream Processing

Mitch Cherniack , Hari Balakrishnan[†], Magdalena Balazinska[†],
Don Carney[‡], U ur Çetintemel[‡], Ying Xing[‡], and Stan Zdonik[‡]

Abstract

Stream processing fits a large class of new applications for which conventional DBMSs fall short. Because many stream-oriented systems are inherently geographically distributed and because distribution offers scalable load management and higher availability, future stream processing systems will operate in a distributed fashion. They will run across the Internet on computers typically owned by multiple cooperating administrative domains. This paper describes the architectural challenges facing the design of large-scale distributed stream processing systems, and discusses novel approaches for addressing load management, high availability, and federated operation issues. We describe two stream processing systems, Aurora* and Medusa, which are being designed to explore complementary solutions to these challenges.

1 Introduction

There is a large class of emerging applications in which data, generated in some external environment, is pushed asynchronously to servers that process this information. Some example applications include sensor networks, location-tracking services, fabrication line management, and network management. These applications are characterized by the need to process high-volume data streams in a timely and responsive fashion. Hereafter, we refer to such applications as *stream-based* applications.

The architecture of current database management systems assumes a *pull-based* model of data access: when a user (the active party) wants data, she submits a query to the system (the passive party) and an answer is returned. In contrast, in stream-based applications data is *pushed* to a system that must evaluate queries in response to detected events. Query answers are then pushed to a waiting user or application. Therefore, the stream-based model inverts the traditional data management model by assuming users to be *passive* and the data management system to be *active.*

Brandeis University, [‡]Brown University, [†]M.I.T.

Proceedings of the 2003 CIDR Conference

Many stream-based applications are naturally distributed. Applications are often embedded in an environment with numerous connected computing devices with heterogeneous capabilities. As data travels from its point of origin (e.g., sensors) downstream to applications, it passes through many computing devices, each of which is a potential target of computation. Furthermore, to cope with time-varying load spikes and changing demand, many servers would be brought to bear on the problem. In both cases, distributed computation is the norm.

This paper discusses the architectural issues facing the design of large-scale distributed stream processing systems. We begin in Section 2 with a brief description of our centralized stream processing system, Aurora [4]. We then discuss two complementary efforts to extend Aurora to a distributed environment: Aurora* and Medusa. Aurora* assumes an environment in which all nodes fall under a single administrative domain. Medusa provides the infrastructure to support federated operation of nodes across administrative boundaries. After describing the architectures of these two systems in Section 3, we consider three design challenges common to both: infrastructures and protocols supporting communication amongst nodes (Section 4), load sharing in response to variable network conditions (Section 5), and high availability in the presence of failures (Section 6). We also discuss high-level policy specifications employed by the two systems in Section 7. For all of these issues, we believe that the push-based nature of stream-based applications not only raises new challenges but also offers the possibility of new domain-specific solutions.

2 Aurora: A Centralized Stream Processor

2.1 System Model

In Aurora, data is assumed to come from a variety of sources such as computer programs that generate values at regular or irregular intervals or hardware *sensors.* We will use the term *data source* for either case. A data *stream* is a potentially unbounded collection of tuples generated by a data source. Unlike the tuples of the relational database model, stream tuples are generated in real-time and are typically not available in their entirety at any given point in time.

Aurora processes tuples from incoming streams according to a specification made by an *application administrator*. Aurora is fundamentally a data-flow system and uses the popular *boxes* and *arrows* paradigm

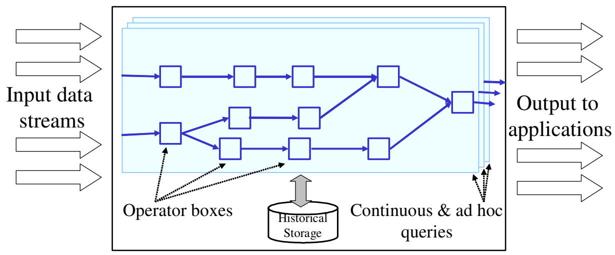

Figure 1: Basic Aurora System Model

found in most process flow and workflow systems. Here, tuples flow through a loop-free, directed graph of processing operators (i.e., *boxes*), as shown in Figure 1. Ultimately, output streams are presented to *applications*, which must be constructed to handle the asynchronously arriving tuples in an output stream.

Every Aurora application must be associated with a *query* that defines its processing requirements, and a *Quality-of-Service* (*QoS*) specification that specifies its performance requirements (see Section 7.1).

2.2 Query Model

Queries are built from a standard set of well-defined operators (*boxes*). Each operator accepts input streams (*in arrows*), transforms them in some way, and produces one or more output streams (*out arrows*). By default, queries are *continuous* [5] in that they can potentially run forever over push-based inputs. *Ad hoc* queries can also be defined and attached to *connection points*: predetermined arcs in the flow graph where historical data is stored.

Aurora queries are constructed using a box-and-arrow based graphical user interface. It would also be possible to allow users to specify declarative queries in a language such as SQL (modified to specify continuous queries), and then compile these queries into our box and arrow representation.

Here, we informally describe a subset of the Aurora operators that are relevant to this paper; a complete description of the operators can be found in [2, 4]. This subset consists of a simple unary operator (Filter), a binary merge operator (Union), a time-bounded windowed sort (WSort), and an aggregation operator (Tumble). Aurora also includes a mapping operator (Map), two additional aggregate operators (XSection and Slide), a join operator (Join), and an extrapolation operator (Resample), none of which are discussed in detail here.

Given some predicate, p, Filter (p) produces an output stream consisting of all tuples in its input stream that satisfy p. Optionally, Filter can also produce a second output stream consisting of those tuples which did not satisfy p. Union produces an output stream consisting of all tuples on its n input streams. Given a set of sort attributes, $A_1, A_2, \ldots, A_n$ and a timeout, WSort buffers all incoming tuples and emits tuples in its buffer in

```
1. (A = 1, B = 2)
2. (A = 1, B = 3)
3. (A = 2, B = 2)
4. (A = 2, B = 1)
5. (A = 2, B = 6)
6. (A = 4, B = 5)
7. (A = 4, B = 2)
        ...
```

Figure 2: A Sample Tuple Stream

ascending order of its sort attributes, with at least one tuple emitted per timeout period.[*]

Tumble takes an input *aggregate function* and a set of input *groupby attributes*.[†] The aggregate function is applied to disjoint "windows" (i.e., tuple subsequences) over the input stream. The groupby attributes are used to map tuples to the windows they belong to. For example, consider the stream of tuples shown in Figure 2. Suppose that a Tumble box is defined with an aggregate function that computes the average value of B, and has A as its *groupby* attribute. This box would emit two tuples and have another tuple computation in progress as a result of processing the seven tuples shown. The first emitted tuple, `(A = 1, Result = 2.5)`, which averages the two tuples with `A = 1` would be emitted upon the arrival of tuple #3: the first tuple to arrive with a value of A not equal to 1. Similarly, a second tuple, `(A = 2, Result = 3.0)`, would be emitted upon the arrival of tuple #6. A third tuple with `A = 4` would not get emitted until a later tuple arrives with A not equal to 4.

2.3 Run-time Operation

The single-node Aurora run-time architecture is shown in Figure 3. The heart of the system is the *scheduler* that determines which box to run. It also determines how many of the tuples that might be waiting in front of a given box to process and how far to push them toward the output. We call this latter determination *train scheduling* [4]. Aurora also has a *Storage Manager* that is used to buffer queues when main memory runs out. This is particularly important for queues at connection points since they can grow quite long.

Aurora must constantly monitor the QoS of output tuples (*QoS Monitor* in Figure 3). This information is important since it drives the *Scheduler* in its decision-making, and it also informs the *Load Shedder* when and where it is appropriate to discard tuples in order to shed load. Load shedding is but one technique employed by Aurora to improve the QoS delivered to applications.

[*] Note that WSort is potentially lossy because it must discard any tuples that arrive after some tuple that follows it in sort order has already been emitted.

[†] Aurora's aggregate operators have two additional parameters that specify when tuples get emitted and when an aggregate times out. For the purposes of this discussion, we assume that these parameters have been set to output a tuple whenever a window is full (i.e., never as a result of a timeout).

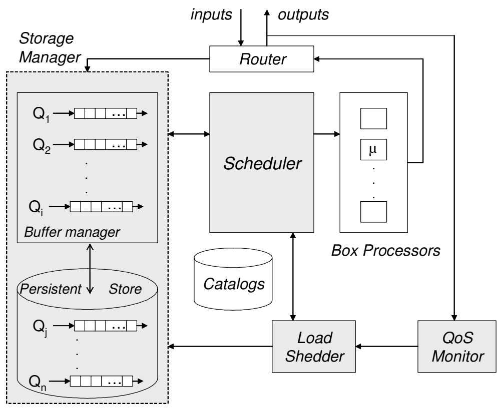

Figure 3: Aurora Run-time Architecture

When load shedding is not working, Aurora will try to re-optimize the network using standard query optimization techniques (such as those that rely on operator commutativities). This tactic requires a more global view of the network and thus is used more sparingly. It does have the advantage that in transforming the original network, it might uncover new opportunities for load shedding. The final tactic is to retune the scheduler by gathering new statistics or switching scheduler disciplines.

3 Distributed System Architecture

Building a large-scale distributed version of a stream processing system such as Aurora raises several important architectural issues. In general, we envision a distributed federation of participating nodes in different administrative domains. Together, these nodes provide a stream processing service for multiple concurrent stream-based applications. Collaboration between distinct administrative domains is fundamentally important for several reasons, including:

1. A federation in which each participating organization contributes a modest amount of computing, communication, and storage resources allows for a high degree of resource multiplexing and sharing, enabling large time-varying load spikes to be handled. It also helps improve fault-tolerance and resilience against denial-of-service attacks.
2. Many streaming services, such as weather forecasting, traffic management, and market analysis, inherently process data from different autonomous domains and compose them; distribution across administrative boundaries is a fundamental constraint in these situations.

We envision that programs will continue to be written in much the same way that they are with single-node Aurora, except that they will now run in a distributed fashion. The partitioning of the query plan on to the participating nodes in response to changing demand, system load, and failures, is a challenging problem, and intractable as an optimization problem in a large network. Additionally, inter-domain collaborations are not a straightforward extension of intra-domain distribution. For instance, some applications may not want their data or computation running within arbitrary domains, and some organizations may not have the incentive to process streams unless they derive tangible benefits from such processing

Our architecture splits the general problem into *intra-participant* distribution (a relatively small-scale distribution all within one administrative domain, handled by Aurora*) and *inter-participant* distribution (a large-scale distribution across administrative boundaries, handled by Medusa). This method of splitting allows the general problem to become tractable, enabling the implementation of different policies and algorithms for load sharing. This decomposition allows three pieces to be shared between Aurora* and Medusa: (i) Aurora, (ii) an overlay network for communication, and (iii) algorithms for high-availability that take advantage of the streaming nature of our problem domain.

3.1 Aurora*: Intra-participant Distribution

Aurora* consists of multiple single-node Aurora servers that belong to the same administrative domain and cooperate to run the Aurora query network on the input streams. In general there are no operational restrictions regarding the nodes where sub-queries can run; boxes can be placed on and executed at arbitrary nodes as deemed appropriate.

When an Aurora query network is first deployed, the Aurora* system will create a crude partitioning of boxes across a network of available nodes, perhaps as simple as running everything on one node. Each Aurora node supporting the running system will continuously monitor its local operation, its workload, and available resources (e.g., CPU, memory, bandwidth, etc.). If a machine finds itself short of resources, it will consider offloading boxes to another appropriate Aurora node. All dynamic reconfiguration will take place in such a decentralized fashion, involving only local, pair-wise interactions between Aurora nodes. We discuss the pertinent load distribution mechanisms and policies in more detail in Section 5.

3.2 Medusa: Inter-participant Federated Operation

Medusa is a distributed infrastructure that provides service delivery among autonomous *participants*. A Medusa participant is a collection of computing devices administered by a single entity. Hence, participants range in scale from collections of stream processing nodes capable of running Aurora and providing part of the global service, to PCs or PDAs that allow user access to the system (e.g., to specify queries), to networks of sensors and their proxies that provide input streams.

Medusa is an *agoric system* [16], using economic principles to regulate participant collaborations and solve the hard problems concerning load management and sharing. Participants provide services to each other by establishing contracts that determine the appropriate compensation for each service. Medusa uses a market mechanism with an underlying currency ("dollars") that backs these contracts. Each contract exists between two participants and covers a message stream that flows between them. One of the contracting participants is the sending participant; the other is the receiving participant. Medusa models each message stream as having positive value, with a well-defined value per message; the model therefore is that the receiving participant always pays the sender for a stream. In turn, the receiver performs query-processing services on the message stream that presumably increases its value, at some cost. The receiver can then sell the resulting stream for a higher price than it paid and make money.

Some Medusa participants are purely stream sources (e.g., sensor networks and their proxies), and are paid for their data, while other participants (e.g., end-users) are strictly stream sinks, and must pay for these streams. However, most Medusa participants are "interior" nodes (acting both as sources and sinks). They are assumed to operate as profit-making entities; i.e., their contracts have to make money or they will cease operation. Our hope is that such contracts (mostly bilateral) will allow the system to *anneal* to a state where the economy is stable, and help derive a practical solution to the computationally intractable general partitioning problem of placing query operators on to nodes. The details of the contracting process are discussed in Section 7.2.

4 Scalable Communications Infrastructure

Both Aurora* and Medusa require a scalable communication infrastructure. This infrastructure must (1) include a naming scheme for participants and query operators and a method for discovering where any portion of a query plan is currently running and what operators are currently in place, (2) route messages between participants and nodes, (3) multiplex messages on to transport-layer streams between participants and nodes, and (4) enable stream processing to be distributed and moved across nodes. The communications infrastructure is an *overlay network*, layered on top of the underlying Internet substrate.

4.1 Naming and Discovery

There is a single global namespace for participants, and each participant has a unique global name. When a participant defines a new operator, schema, or stream, it does so within its own namespace. Hence, each entity's name begins with the name of the participant who defined it, and each object can be uniquely named by the tuple: (*participant*, *entity-name)*. Additionally, a stream that crosses participant boundaries is named separately within each participant.

To find the definition of an entity given its name, or the location where a data stream is available or a piece of a query is executing, we define two types of catalogs in our distributed infrastructure: intra-participant and inter-participant catalogs. Within a participant, the catalog contains definitions of operators, schemas, streams, queries, and contracts. For streams, the catalog also holds (possibly stale) information on the physical locations where events are being made available. Indeed, streams may be partitioned across several nodes for load balancing. For queries, the catalog holds information on the content and location of each running piece of the query. The catalog may be centralized or distributed. All nodes owned by a participant have access to the complete intra-participant catalog.

For participants to collaborate and offer services that cross their boundaries, some information must be made globally available. This information is stored in an inter-participant catalog and includes the list, description, and current location of pieces of queries running at each participant.

Each participant that provides query capabilities holds a part of the shared catalog. We propose to implement such a distributed catalog using a distributed hash table (DHT) with entity names as unique keys. Several algorithms exist for this purpose (e.g., DHTs based on consistent hashing [6, 14] and LH* [19]). These algorithms differ in the way they distribute load among participants, handle failures, and perform lookups. However, they all efficiently locate nodes for any key-value binding, and scale with the number of nodes and the number of objects in the table.

4.2 Routing

Before producing events, a data source, or an administrator acting on its behalf, registers a new schema definition and a new stream name with the system, which in turn assigns a default location for events of the new type. Load sharing between nodes may later move or partition the data. However, the location information is always propagated to the intra-participant catalog.

When a data source produces events, it labels them with a stream name and sends them to one of the nodes in the overlay network. Upon receiving these events, the node consults the intra-participant catalog and forwards events to the appropriate locations.

Each Aurora network deployed in the system explicitly binds its inputs and outputs to a list of streams, by enumerating their names. When an input is bound, the intra-participant catalog is consulted to determine where the streams of interest are currently located. Events from these streams are then continually routed to the location where the query executes.

Query plans only bind themselves to streams defined within a participant. Explicit connections are opened for streams to cross participant boundaries. These streams are then defined separately within each domain.

4.3 Message Transport

When a node transfers streams of messages to another node in the overlay, those streams will in general belong to different applications and have different characteristics. In many situations, especially in the wide-area, we expect the network to be the stream bottleneck. The transport mechanism between nodes must therefore be carefully designed.

One approach would be to set up individual TCP connections, one per message stream, between the node pair. This approach, although simple to implement, has several problems. First, as the number of message streams grows, the overhead of running several TCP connections becomes prohibitive on the nodes. Second, independent TCP connections do not share bandwidth well and in fact adversely interact with each other in the network [11]. Third, both within one participant as well as between participants, we would like the bandwidth between the nodes to be *shared* amongst the different streams according to a prescribed set of weights that depend on either QoS specifications or contractual obligations.

Our transport approach is to *multiplex* all the message streams on to a single TCP connection and have a message scheduler that determines which message stream gets to use the connection at any time. This scheduler implements a weighted connection sharing policy based on QoS or contract specification, and keeps track of the rates allocated to the different messages in time.

There are some message streaming applications where the in-order reliable transport abstraction of TCP is not needed, and some message loss is tolerable. We plan to investigate if a UDP-based multiplexing protocol is also required in addition to TCP. Doing this would require a congestion control protocol to be implemented [12].

4.4 Remote Definition

To share load dynamically between nodes within a participant, or across participants, parts of Aurora networks must be able to change the location where they execute at run-time. However, process migration raises many intractable compatibility and security issues, especially if the movement crosses participant boundaries. Therefore, we propose a different approach, which we call *remote definition.* With this approach, a participant instantiates and composes operators from a pre-defined set offered by another participant to mimic box sliding. For example, instead of moving a WSort box, a participant remotely defines the WSort box at another participant and binds it to the appropriate streams within the new domain. Load sharing and box sliding are discussed in more details in the following sections.

In addition to facilitating box sliding, remote definition also helps content customization. For example, a participant might offer streams of events indicating stock quotes. A receiving participant interested only in knowing when a specific stock passes above a certain threshold would normally have to receive the complete stream and would have to apply the filter itself. With remote definition, it can instead remotely define the filter, and receive directly the customized content.

5 Load Management

To adequately address the performance needs of stream-based applications under time varying, unpredictable input rates, a multi-node data stream processing system must be able to dynamically adjust the allocation of processing among the participant nodes. This decision will primarily consider the loads and available resources (e.g., processor cycles, bandwidth, memory).

Both Aurora* and Medusa address such load management issues by means of a set of algorithms that provide efficient load sharing among nodes. Because Aurora* assumes that the participants are all under a common administrative control, lightly-loaded nodes will freely share load with their over-burdened peers. Medusa will make use of the Aurora* mechanisms where appropriate, but it must also worry about issues of how to cross administrative boundaries in an economically viable way without violating contractual constraints.

In the rest of this section, we first discuss the basic mechanisms used for partitioning and distributing Aurora operator networks across multiple nodes. We then discuss several key questions that need to be addressed by any repartitioning policy.

5.1 Mechanisms: Repartitioning Aurora Networks

On every node that runs a piece of Aurora network, a query optimizer/load share daemon will run periodically in the background. The main task of this daemon will be to adjust the load of its host node in order to optimize the overall performance of the system. It will achieve this by either off-loading computation or accepting additional computation. Load redistribution is thus a process of moving pieces of the Aurora network from one machine to another.

Load sharing must occur while the network is operating. Therefore, it must first stabilize the network at the point of the transformation. Network transformations are only considered between connection points. Consider a sub-network S that is bounded on the input side by an arc, C_{in}, and on the output side by an arc, C_{out}. The connection point at C_{in} is first choked off by simply collecting any subsequent input arriving at the connection point at C_{in}. Any tuples that are queued within S are allowed to drain off. When S is empty, the network is manipulated, parts of it are moved to other machines, and the flow of messages at C_{in} is turned back on.

It should be noted that the reconfiguration of the Aurora network will not always be a local decision. For example, an upstream node might be required to signal a downstream node that it does not have sufficient

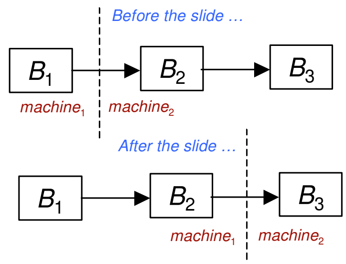

Figure 4: Box Sliding

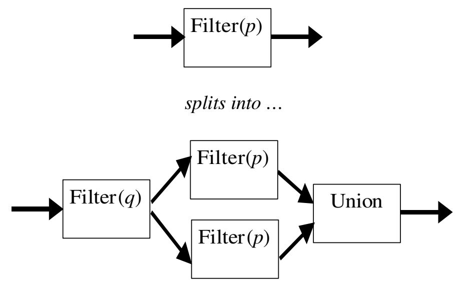

Figure 5: Splitting a Filter Box

bandwidth to handle its output (this would happen if an upstream node notices a backup on its output link). In this case, the upstream node might want to signal the neighboring downstream node to move one or more boxes upstream to reduce the communication across that link.

We now discuss two basic load sharing mechanisms, *box sliding* and *box splitting*, which are used to repartition the Aurora network in a pair-wise fashion.

Box Sliding. This technique takes a box on the edge of a sub-network on one machine and shifts it to its neighbor. Beyond the obvious repositioning of processing, shifting a box upstream is often useful if the box has a low selectivity (reduces the amount of data) and the bandwidth of the connection is limited. Shifting a box downstream can be useful if the selectivity of the box is greater than one (produces more data than the input, e.g., a join) and the bandwidth of the connection is again limited. We call this kind of remapping *horizontal load sharing* or *box sliding*. Figure 4 illustrates upstream box sliding.

It should be noted that the machine to which a box is sent must have the capability to execute the given operation. In a sensor network, some of the nodes can be very weak. Often the sensor itself is capable of computation, but this capability is limited. Thus, it might be possible to slide a simple Filter box to a sensor node, whereas the sensor might not support a Tumble box.

It should also be noted that box sliding could also move boxes vertically. That is, a box that is assigned to machine A can be moved to machine B as long as the input and output arcs are rerouted accordingly.

Box Splitting. A heavier form of load sharing involves *splitting* Aurora boxes. A split creates a copy of a box that is intended to run on a second machine. This mechanism can be used to offload from an overloaded machine; one or more boxes on this machine get split, and some of the load then gets diverted to the box copies resulting from the split (and situated on other machines). Every box-split must be preceded by a Filter box with a predicate that partitions input tuples (routing them to one box or the other). For splits to be transparent (i.e., to ensure that a split box returns the same result as an unsplit box), one or more boxes must be added to the network that merges the box outputs back into a single stream.

The boxes required to merge results depend on the box that is split. Figure 5 and Figure 6 show two examples. The first split is of Filter and simply requires a Union box to accomplish the merge. The second split is of Tumble, which requires a more sophisticated merge, consisting of Union followed by WSort and then another Tumble. It also requires that the aggregate function argument to Tumble, *agg*, have a corresponding combination function, *combine*, such that for any set of tuples, $\{x_1, x_2, \ldots, x_n\}$, and k n:

$$agg(\{x_1, x_2, \ldots, x_n\}) =$$
$$combine(agg(\{x_1, x_2, \ldots, x_k\}), agg(\{x_{k+1}, x_{k+2}, \ldots, x_n\}))$$

For example, if *agg* is *cnt* (count), *combine* is *sum*, and if *agg* is *max*, then *combine* is *max* also. In Figure 6, *agg* is *cnt* and *combine* is *sum*.

To illustrate the split shown in Figure 6, consider a Tumble applied to the stream that was shown in Figure 2 with the aggregate function *cnt* and groupby attribute A. Observe that without splitting, Tumble would emit the following tuples while processing the seven tuples shown in Figure 2:

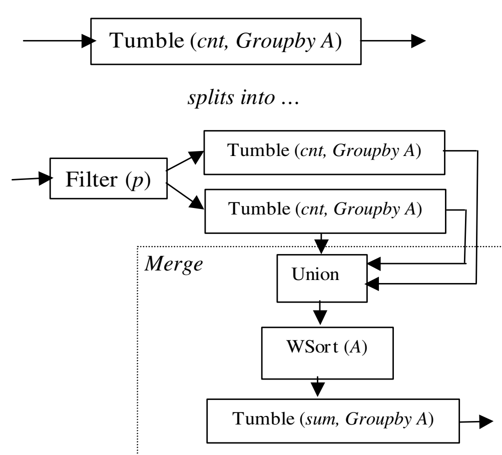

Figure 6: Splitting a Tumble Box

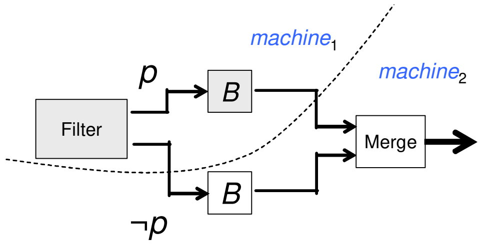

Figure 7: Remapping after a Split

```
(A = 1, result = 2)
(A = 2, result = 3)
```

Suppose that a split of the Tumble box takes place after tuple #3 arrives, and that the Filter box used for routing tuples after the split uses the predicate, `B < 3` to decide where to send any tuple arriving in the future (i.e., if `B < 3` then send the tuple to machine containing the original Tumble box (machine #1), and otherwise send the tuple to machine #2). In this case, machine #1 will see tuples 1, 2, 3, 4 and 7; and machine #2 will see tuples 5 and 6. After machine #1 processes tuple #7, its Tumble box will have emitted tuples:

```
(A = 1, result = 2)
(A = 2, result = 2)
```

and after machine #2 processes tuple #6, its Tumble box will have emitted the tuple:

```
(A = 2, result = 1)
```

Assuming a large enough timeout argument, WSort rearranges the union of these tuples, emitting them in order of their values for A. The Tumble box that follows then adds the values of `result` for tuples with like values of A. This results in the emission of tuples:

```
(A = 1, result = 2)
(A = 2, result = 3)
```

which is identical to that of the unsplit Tumble box.

Once split has replicated a part of the network, the parallel branches can be mapped to different machines. In fact, an overloaded machine may perform a split and then ask a neighbor if it can accept some additional load. If the neighbor is willing, the network might get remapped as in Figure 7.

5.2 Key Repartitioning Challenges

We now provide an outline of several fundamental policy decisions regarding when and how to use the load sharing mechanism described in the previous subsection. Particular solutions will be guided and constrained by the high-level policy specifications and guidelines, QoS and economic contracts, used by Aurora* and Medusa, respectively (see Section 7).

Initiation of Load Sharing. Because network topologies and loads will be changing frequently, load sharing will need to be performed fairly frequently as well. However, shifting boxes around too frequently could lead to instability as the system tries to adjust to load fluctuations. Determining the proper granularity for this operation is an important consideration for a successful system.

Choosing What to Offload. Both box sliding and box splitting require moving boxes and their input and output arcs across machine boundaries. Even though a neighboring machine may have available compute cycles and memory, it may not be able to handle the additional bandwidth of the new arcs. Thus, the decision of which Aurora network pieces to move must consider bandwidth availability as well.

Choosing Filter Predicates for Box Splitting. Every box split results in a new sub-network rooted by a Filter box. The Filter box acts as a semantic router for the tuples arriving at the box that has been split. The filter predicate, *p*, defines the redistributed load. The choice of *p* is crucial to the effectiveness of this strategy. Predicate *p* could depend on the stream content. For example, we might want to separate streams based on where they were generated as in *all streams generated in Cambridge*. On the other hand, the partitioning criterion could depend on some metadata or statistics about the streams as in *the top 10 streams by arrival rate*. Alternatively, *p* could be based on a simple statistic as in *half of the available streams*. Moreover, the choice of *p* could vary with time. In other words, as the network characteristics change, a simple adjustment to *p* could be enough to rebalance the load.

Choosing What to Split. Choosing the right sub-network to split is also an important optimization problem. The trick is to pick a set of boxes that will move "just enough" processing. In a large Aurora network, this could be quite difficult. Moreover, it is important to move load in way that will not require us to move it again in the near future. Thus, finding candidate sub-networks that have durable effect is important.

Handling Connection Points. Naively, splitting a connection point could involve copying a lot of data. Depending on the expected usage, this might be a good investment. In particular, if it is expected that many users will attach ad hoc queries to this connection point, then splitting it and moving a replica to different machine may be a sensible load sharing strategy. On the other hand, it might make sense to leave the connection point intact and to split the boxes on either side of it. This would mean that the load introduced by the processing would be moved, while the data access to the second box would be remote.

6 High Availability

A key goal in the design of any data stream processing system is to achieve robust operation in volatile and dynamic environments, where availability may suffer due to (1) server and communication failures, (2) sustained congestion levels, and (3) software failures. In order to improve overall system availability, Aurora* and Medusa rely on a common stream-oriented data back-up and recovery approach, which we describe below.

6.1 Overview and Key Features

Our high-availability approach has two unique advantages, both due to the streaming data-flow nature of our target systems. First, it is possible to reliably back up data and provide safety without incurring the overhead to explicitly copy them to special back up servers (as in the case of traditional process pair models [10]). In our model, each server can effectively act as a back-up for its downstream servers. Tuples get processed and flow naturally in the network (precisely as in the case of regular operation). Unlike in regular operation, however, processed tuples are discarded *lazily*, only when it is determined that their *effects* are safely recorded elsewhere, and, thus, can be effectively recovered in case of a failure.

Second, the proposed approach enables a tradeoff between the recovery time and the volume of checkpoint messages required to provide safety. This flexibility allows us to emulate a wide spectrum of recovery models, ranging from a high-volume checkpoints/fast-recovery approach (e.g., Tandem [1]) to a low-volume checkpoints/slow-recovery approach (e.g., log-based recovery in traditional databases).

6.2 Regular Operation

We say that a distributed stream processing system is *k-safe* if the failure of any *k* servers does not result in any message losses. The value of *k* should be set based on the availability requirements of applications, and the reliability and load characteristics of the target environments. We provide *k-safety* by maintaining the copies of the tuples that are *in transit* at each server *s*, at *k* other servers that are upstream from *s*. An upstream back-up server simply holds on to a tuple it has processed until its primary server tells it to discard the tuple. Figure 8 illustrates the basic mechanism for $k = 1$. Server s_1 acts as a back-up of server s_2. A tuple *t* sent from s_1 to s_2 is simply kept at s_1's output queue until it is guaranteed that all tuples that *depended* on *t* (i.e., the tuples whose values got determined directly or indirectly based on *t*) made it to s_3.

In order to correctly truncate output queues, we need to keep track of the order in which tuples are transmitted

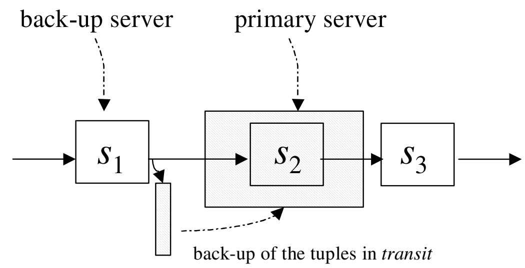

Figure 8: Primary and back-up servers

between the servers. When an upstream server sends a message (containing tuples) to a successor, it also includes a monotonically increasing sequence number. It is sufficient to include only the base sequence number, as the corresponding numbers for all tuples can be automatically generated at the receiving server by simply incrementing the base. We now describe two remote queue truncation techniques that use tuple sequence numbers.

Our first technique involves the use of special *flow messages*. Periodically, each data source creates and sends flow messages into the system. A box processes a flow message by first recording the sequence number of the earliest tuple that it currently depends on[‡], and then passing it onward. Note that there might be multiple earliest sequence numbers, one for each upstream server at the extreme case. When the flow message reaches a server boundary, these sequence values are recorded and the message continues in the next server. Hence, each server records the identifiers of the earliest upstream tuples that it depends on. These values serve as checkpoints; they are communicated through a *back channel* to the upstream servers, which can appropriately truncate the tuples they hold. Clearly, the flow message can also be piggybacked on other control or data messages (such as heartbeat messages, DHT lookup messages, or regular tuple messages).

The above scheme will operate correctly only for *straight-line* networks. When there are branches and recombinations, special care must be taken. Also, when messages from one server go to multiple subsequent servers, additional extensions are required.

Whenever a message is split and sent to two destinations, then the flow message is similarly split. If a box gets input from two arcs, it must save the first flow message until it receives one on the other arc. If the two flow messages come from different servers, then both are sent onward. If they come from the same server, then the

[‡] If the box has state (e.g., consider an aggregate box), then the recorded tuple is the one that presently contributes to the state of the box and that has the lowest sequence number (for each upstream server). If the box is stateless (e.g., a filter box), then the recorded tuple is the one that has been processed most recently.

minimum is computed as before and a single message sent onward. In this way, the correct minimum is received at the output.

An output can receive flow messages from multiple upstream servers. It must merely respond to the correct one with a back channel message. Similarly, when an upstream server has multiple successor servers, it must wait for a back channel message from each one, and then only truncate the queue to the maximum of the minimum values.

An alternate technique to special flow messages is to install an array of sequence numbers on each server, one for each upstream server. On each box's activation, the box records in this array the earliest tuples on which it depends. The upstream servers can then query this array periodically and truncate their queues accordingly. This approach has the advantage that the upstream server can truncate at its convenience, and not just when it receives a back channel message. However, the array approach makes the implementation of individual boxes somewhat more complex.

6.3 Failure Detection and Recovery

Each server sends periodic heartbeat messages to its upstream neighbors. If a server does not hear from its downstream neighbor for some predetermined time period, it considers that its neighbor failed, and it initiates a recovery procedure. In the recovery phase, the back-up server itself immediately starts processing the tuples in its output log, emulating the processing of the failed server for the tuples that were still being processed at the failed server. Subsequently, load-sharing techniques can be used to offload work from the back-up server to other available servers. Alternatively, the backup server can move its output log to another server, which then takes over the processing of the failed server. This approach might be worthwhile if the back-up server is already heavily loaded and/or migration of the output log is expected to be inexpensive.

6.4 Recovery Time vs. Back Up Granularity

The above scheme does not interfere with the natural flow of tuples in the network, providing high availability with only a minimum of extra messages. In contrast, a process-pair approach requires check pointing a computation to its backup on a regular basis. To achieve high availability with a process-pair model would require a checkpoint message every time a box processed a message. This is overwhelmingly more expensive than the approach we presented. However, the cost of our scheme is the possibly considerable amount of computation required during recovery. In contrast, a process-pair scheme will redo only those box calculations that were in process at the time of the failure. Hence, the proposed approach saves many run-time messages, at the expense of having to perform additional work at failover time.

The basic approach can be extended to support faster recovery, but at higher run time cost. Consider establishing a collection of K virtual machines on top of the Aurora network running on a single physical server. Now, utilize the approach described above *for each virtual machine*. Hence, there will be queues at each virtual machine boundary, which will be truncated when possible. Since each queue is on the same physical hardware as its downstream boxes, high availability is not provided on machine failures with the algorithms described so far.

To achieve high-availability, the queue has to be replicated to a physical backup machine. At a cost of one message per entry in the queue, each of the K virtual machines can resume processing from its queue, and finer granularity restart is supported. The ultimate extreme is to have one virtual machine per box. In this case, a message must be sent to a backup server each time a box processes a message. However, only the processing of the in-transit boxes will be lost. This will be very similar to the process-pair approach. Hence, by adding virtual machines to the high-availability algorithms, we can tune the algorithms to any desired tradeoff between recovery time and run time overhead.

7 Policy Specifications and Guidelines

We now describe the high-level policy specifications employed by Aurora* and Medusa to guide all pertinent resource, load, and availability management decisions. We first describe application-specific QoS specifications used by Aurora*, and then overlay the Medusa approach for establishing (economic) contracts between different domains.

7.1 QoS Based Control in Aurora*

Along with a query, every Aurora application must also specify its QoS expectations [4]. A QoS specification is a function of some *performance, result precision, or reliability* related characteristic of an output stream that produces a utility (or *happiness*) value to the corresponding application. The operational goal of Aurora is to maximize the perceived aggregate QoS delivered to the client applications. As a result, all Aurora resource allocation decisions, such as scheduling and load shedding, are driven by QoS-aware algorithms [4]. We now discuss some interesting issues that arise as we extend the basic single-node QoS model to a distributed multi-node model to be used by Aurora*.

One key QoS issue that needs to be dealt with in Aurora* involves *inferring* QoS for the outputs of arbitrary Aurora* nodes. In order to be consistent with the basic Aurora model and to minimize the coordination among the individual Aurora nodes, it is desirable for each node in an Aurora* configuration to run its own local Aurora server. This requires the presence of QoS specifications at the outputs of *internal* nodes (i.e., those that are not directly connected to output applications).

Because QoS expectations are defined only at the output nodes, the corresponding specifications for the internal nodes must be properly inferred. This inference is illustrated in Figure 9, where a given application's query result is returned by node S_3, but additional computation is done at the internal nodes S_1 and S_2. The QoS specified at the output node S_3 needs to be pushed inside the network, to the outputs of S_1 and S_2, so that these internal nodes can make local resource management decisions.

While, in general, inferring accurate QoS requirements in the middle of an Aurora network is not going to be possible, we believe that inferring good approximations to some of the QoS specifications (such as the latency-based QoS specification, which is a primary driver for many resource control issues) is achievable given the availability of operational system statistics. To do this, we assume that the system has access to the average processing cost and the selectivity of each box. These statistics can be monitored and maintained in an approximate fashion over a running network.

A QoS specification at the output of some box, B is a function of time t and can be written as $Q_o(t)$. Assume that box B takes, on average, T_B units of time for a tuple arriving at its input to be processed completely. T_B can be measured and recorded by each box and would implicitly include any queuing time. The QoS specification $Q_i(t)$ at box B's input would be $Q_o(t+T_B)$. This simple technique can be applied across an arbitrary number of Aurora boxes to compute an estimated latency graph for any arc in the system.

Another important issue relates to the *precision* (i.e., accuracy) of query results. Precise answers to queries are sometimes unachievable or undesirable, both of which potentially lead to dropped tuples. A *precise query answer* is what would be returned if no data was ever dropped, and query execution could complete regardless of the time it required. A precise query answer might be unachievable (from an Aurora system's perspective) if high load on an Aurora server necessitated dropping tuples. A precise query answer might be undesirable (from an Aurora application's perspective) if a query depended upon data arriving on an extremely slow stream, and an approximate but fast query answer was preferable to one that was precise but slow. QoS specifications describe, from an applications' perspective, what measures that it prefer Aurora take under such circumstances. For example, if tuples must be dropped, QoS specifications can be used to determine which and how many.

Because imprecise query answers are sometimes unavoidable or even preferable to precise query answers, *precision* is the wrong standard for Aurora systems to strive for. In general, there will be a continuum of acceptable answers to a query, each of which has some measurable deviation from the perfect answer. The degree of tolerable approximation is application specific; QoS specifications serve to define what is acceptable.

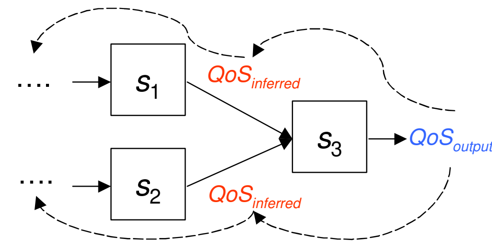

Figure 9: Inferring QoS at intermediate nodes

7.2 Economic Contract Based Control in Medusa

As discussed in previous sections, Medusa regulates interactions between participants using an agoric model with three basic types of contracts: (a) content contracts (b) suggested contracts, and (c) movement contracts. We discuss each type of contract in turn.

Content contracts cover the payment by a receiving participant for the stream to be sent by a sending participant. The form of a content contract is:

For stream_name
For time period
With availability guarantee
Pay payment

Here, stream_name is a stream known to the sender, which the receiver must map to a local stream name. The time period is the amount of time that sender will make the stream available to the receiver, and payment is the amount of money remitted. Payment can either be a fixed dollar amount (subscription) or it can be a per-message amount. An optional availability clause can be added to specify the amount of outage that can be tolerated, as a guarantee on the fraction of uptime.

With content contracts, Medusa participants can perform services for each other. Additionally, if participants authorize each other to do remote definitions, then buying participants can easily customize the content that they buy by defining a query plan at the selling participant. These two types of interactions form the basis of our system.

Additional contracts are needed to manage load among participants and optimize queries. For instance, participant P can use remote definition and content contracts to partition a query plan Q over a set of other participants $\{P_1, \ldots, P_k\}$ in an arbitrary manner. P needs to have remote definition authorization at each of P_1 through P_k, but the latter do not need to have contracts with each other. Unfortunately, this form of collaboration will require that query plans be "star shaped" with P in the middle, since P_1 through P_k don't have contractual relationships with each other.

To facilitate more efficient plans and inter-participant load management, we need the ability to modify the way queries are partitioned across participants at run time.

More precisely, we need the ability to slide boxes across participants as well as the ability to add or remove a participant from a query-processing path. For instance, we would like to remove P from the star-shaped query defined above.

Adding a participant to a query plan is straightforward with remote definition and content contracts. Removing a participant requires that the leaving participant ask other participants to establish new content contracts with each other. The mechanism for this is suggested contracts: a participant P suggests to downstream participants an alternate location (participant and stream name) from where they should buy content currently provided by P. Receiving participants may ignore suggested contracts.

The last form of contract facilitates load balancing via a form of box sliding, and is called a movement contract. Using remote definition, a participant P_1 defines a query plan at another participant, P_2. Using a content contract, this remote query plan can be activated. To facilitate load balancing, P_1 can define not one, but a set of L remote query plans. Paired with locally running queries (upstream or downstream), these plans provide equivalent functionality, but distribute load differently across P_1 and P_2. Hence, a movement contract between two participants contains a set of distributed query plans and corresponding inactive content contracts. There is a separate movement contract for each query crossing the boundary between two participants. An oracle on each side determines at runtime whether a query plan and corresponding content contracts from one of the movement contracts is preferred to any of currently active query plans and content contracts. If so, it communicates with the counterpart oracle to suggest a substitution; i.e., to make the alternate query plan (and its corresponding content contracts) active instead of the current query plan and contracts. If the second oracle agrees, then the switch is made. In this way, two oracles can agree to switch query plans from time to time.

A movement contract can be cancelled at any time by either of the participants. If a contract is cancelled and the two oracles do not agree on a replacement, then co-operation between the two participants reverts to the existing content contract (if one is in place). Hence movement contracts can be used for dynamic load balancing purposes. Of course, oracles must carefully monitor local load conditions, and be aware of the economic model that drives contracting decisions at the participant. Additionally, in the same manner as content contracts, movement contracts can also be transferred using suggested contracts.

8 Related Work

We now briefly discuss previous related research, focusing primarily on load sharing, high availability, and distributed mechanisms for federated operations.

Load sharing has been extensively studied in a variety of settings, including distributed operating systems (e.g., [9, 20]) and databases (e.g., [3, 7]). In a distributed system, the load typically consists of multiple independent tasks (or processes), which are the smallest logical units of processing. In Aurora, the corresponding smallest processing units are individual operators that exhibit input-output dependencies, complicating their physical distribution.

Several distributed systems [8, 17] investigated on-the-fly task *migration* and *cloning* as means for dynamic load sharing. Our *Slide* and *Split* operations not only facilitate similar (but finer-grained) load sharing, but also take into account operator dependencies mentioned above, properly splitting and merging the input and resulting data streams as necessary.

Parallel database systems [3, 7] typically share load by using operator splitting and data partitioning. Since Aurora operators are stream-based, the details of how we split the load and merge results are different. More importantly, existing parallel database query execution models are relatively static compared to Aurora* and Medusa: they do not address continuous query execution, and as a result, do not also consider adaptation issues.

Because our load sharing techniques involve dynamically transforming query plans, systems that employ dynamic query optimization are also relevant (e.g., see [13] for a survey). These system change query plans on the fly in order to minimize query execution cost, reduce query response time, or maximize output rates; whereas our motivation is to enable dynamic cross-machine load distribution. Furthermore, most dynamic query optimization research addressed only centralized query processing. The ones that addressed distributed/parallel execution relied on centralized query optimization and load sharing models. Our mechanisms and policies, on the other hand, implement dynamic query re-configuration and load sharing in a truly decentralized way in order to achieve high scalability.

While we have compared our back-up and recovery approach with the generic process-pair model in Section 4, a variation of this model [15] provides different levels of availability for workflow management systems. Instead of backing up process states, the system logs changes to the workflow *components*, which store inter-process messages. This approach is similar to that of ours, in that system state can be recovered by reprocessing the component back-ups. Unlike our approach, however, this approach does not take advantage of the data-flow nature of processing, and therefore has to explicitly back up the components at remote servers.

Market-based approaches rely on economic principles to value available resources and match supply and demand. Mariposa [18] is a distributed database system that uses economic principles to guide data management decisions. While Mariposa's resource pricing and trade are on a query-by-query basis, the trade in Medusa is based on service subscriptions. Medusa contracts enable participants to collaborate and share load with reasonably low overhead, unlike Mariposa's per-query bidding process.

9 Conclusions

This paper discusses architectural issues encountered in the design of two complementary large-scale distributed stream processing systems. Aurora* is a distributed version of the stream processing system Aurora, which assumes nodes belonging to a common administrative domain. Medusa is an infrastructure supporting the federated operation of several Aurora nodes across administrative boundaries. We discussed three design goals in particular: a scalable communication infrastructure, adaptive load management, and high availability. For each we discussed mechanisms for achieving these goals, as well as policies for employing these mechanisms in both the single domain (Aurora*) and federated (Medusa) environments. In so doing, we identified the key challenges that must be addressed and opportunities that can be exploited in building scalable, highly available distributed stream processing systems.

Acknowledgments

We are grateful to Mike Stonebraker for his extensive contributions to the work described in this paper. Funding for this work at Brandeis and Brown comes from NSF under grant number IIS00-86057, and at MIT comes from NSF ITR ANI-0205445.

References

[1] Tandem Database Group. Non-Stop SQL: A Distributed, High Performance, High-Reliability Implementation of SQL. In *Proceedings of the Workshop on High Performance Transaction Systems*, Asilomar, CA, 1987.

[2] D. Abbadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. Brown Computer Science CS-02-10, August 2002.

[3] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic Load Balancing in Hierarchical Parallel Database Systems. In *Proceedings of International Conference on Very Large Data Bases*, Bombay, India, 1996.

[4] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, 2002.

[5] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, TX, 2000.

[6] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 1997.

[7] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85-98, 1992.

[8] F. Douglis and J. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International IEEE Conference on Distributed Computing Systems*, Berlin, Germany, 1987.

[9] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662-675, 1986.

[10] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*: Morgan Kaufman, 1993.

[11] H. Balakrishnan, H. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *ACM SIGCOMM*, Cambridge, MA, 1999.

[12] H. Balakrishnan and S. Seshan. The Congestion Manager, RFC 3124.

[13] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23(2):7-18, 2000.

[14] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, San Diego, CA, 2001.

[15] M. Kamath, G. Alonso, R. Guenthor, and C. Mohan. Providing High Availability in Very Large Workflow Management Systems. In *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, 1996.

[16] M.S. Miller and K. E. Drexler, "Markets and Computation: Agoric Open Systems," in *The Ecology of Computation*, B. A. Huberman, Ed.: North-Holland, 1988.

[17] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys*, 32(241-299, 2000.

[18] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal: Very Large Data Bases*, 5(1):48-63, 1996.

[19] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* - A Scalable Distributed Data Structure. *ACM Transactions on Data Base Systems*, 21(4):480-525, 1996.

[20] W. Zhu, C. Steketee, and B. Muilwijk. Load balancing and workstation autonomy on Amoeba. *Australian Computer Science Communications*, 17(1):588--597, 1995.