Distributed Operation in the Borealis Stream Processing Engine *

Yanif Ahmad yna@cs.brown.edu
Bradley Berg bb@cs.brown.edu
Uğur Çetintemel ugur@cs.brown.edu
Mark Humphrey msh@cs.brown.edu
Jeong-Hyon Hwang jhhwang@cs.brown.edu
Anjali Jhingran anjali@cs.brown.edu
Anurag Maskey† anurag@cs.brandeis.edu
Olga Papaemmanouil olga@cs.brown.edu
Alexander Rasin alexr@cs.brown.edu
Nesime Tatbul tatbul@cs.brown.edu
Wenjuan Xing vivian@cs.brown.edu
Ying Xing yx@cs.brown.edu
Stan Zdonik sbz@cs.brown.edu

Brown University, Providence, RI. †Brandeis University, Waltham, MA.

ABSTRACT

Borealis is a distributed stream processing engine that is being developed at Brandeis University, Brown University, and MIT. Borealis inherits core stream processing functionality from Aurora and inter-node communication functionality from Medusa.

We propose to demonstrate some of the key aspects of distributed operation in Borealis, using a multi-player network game as the underlying application. The demonstration will illustrate the dynamic resource management, query optimization and high availability mechanisms employed by Borealis, using visual performance-monitoring tools as well as the gaming experience.

1. INTRODUCTION

Over the last several years, a great deal of research has been accomplished in the area of stream processing engines. Several groups have developed working prototypes [1, 5, 10] and many papers have been published on detailed aspects of the technology (e.g., [3, 7]).

Borealis is a distributed stream processing engine (SPE) that inherits core stream processing functionality from Aurora [5] and inter-node communication functionality from Medusa [13]. The Borealis design is driven by our experience in using Aurora and Medusa, in developing several streaming applications including the Linear Road benchmark [2], and in pursuing commercial opportunities. Borealis modifies and extends both systems in nontrivial and critical ways to provide advanced capabilities that are commonly required by newly emerging stream processing applications. In particular, Borealis extends the basic Aurora system with the ability to (1) modify various data and query attributes at run time, in an undisruptive manner, and (2) operate in a distributed fashion.

*This work has been supported in part by the National Science Foundation under the ITR grants IIS-0325838 and IIS-0325525.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

In this demonstration, we focus on the key aspects of distributed operation in Borealis. Other unique features (such as "revision records", "time travel", and "control lines" that enable the first capability described above) are described elsewhere [6].

Key reasons for distributing stream processing across multiple machines include:

- **Incremental scalability**: The system can scale up and deal with increasing load or time-varying load spikes, with the addition of new computational resources.
- **High availability**: Multiple processing nodes can monitor system health and perform fast fail-over and recovery in the case of failures.

2. BOREALIS ARCHITECTURE

Borealis is a distributed stream processing engine [6]. The collection of continuous queries submitted to Borealis can be seen as one giant network of operators (a.k.a. query diagram) whose processing is distributed across multiple sites.

Each Borealis site runs a server whose major components are shown in Figure 1. Query execution occurs locally within the *Query Processor (QP)* module. Borealis *I/O Queues* feed input data into the QP and route tuples between remote Borealis nodes and clients.

The *Admin* module is responsible for controlling the local QP, performing tasks such as setting up queries, and migrating diagram fragments. The Admin module coordinates with our *Local Optimizer (LOpt)* to find performance enhancements. LOpt employs scheduling techniques, modification of boxes' runtime characteristics, and our *Load Shedder*, which discards low-priority tuples when the node is overloaded.

Other than the QP, a Borealis node has modules which communicate with their peers on other Borealis nodes to take collaborative actions. The *Neighborhood Optimizer (NOpt)* uses local load information as well as information from other NOpts to improve load balance between nodes. The *High Availability (HA)* modules on different nodes monitor each other and take over processing for one another in case of failure. *Local Monitor* collects performance-related statistics as the local system runs to report to local and neighborhood optimizer modules. The *Global Catalog* provides access to a single logical representation of the diagram.

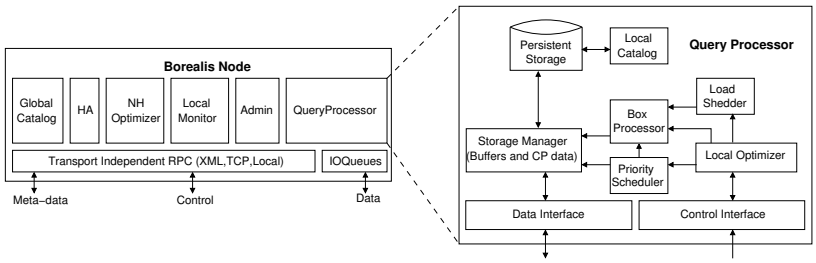

Figure 1: Borealis Architecture

3. DEMONSTRATION FEATURES

3.1 Load Distribution

Borealis has a correlation-based load distributor [12] that distributes the workload on Borealis servers dynamically. The load distributor is designed to cope with fluctuating and bursty workloads. The basic goal is to minimize the average end-to-end processing latency.

The load distribution algorithm differs from the traditional load balancing algorithms in that it not only balances the average load of the nodes, but also attempts to (1) minimize the average load variation on the nodes and (2) maximize the average load correlation of the node pairs. The first goal helps in minimizing the data queueing latencies and the second helps in minimizing the number of load migrations needed and thus, latencies resulting from operator migration.

Our load distributor functions as either a global algorithm or a pair-wise operator exchange algorithm. The global algorithm allocates all operators on all nodes at the same time, for example to initially distribute operators. The pair-wise algorithm redistributes operators between selected node pairs and is used for dynamic load redistribution.

3.2 Load Shedding

The load shedder is responsible for detecting and handling overload situations [9]. Load shedding is accomplished by shedding tuples, by temporarily adding "drop" operators to the Borealis processing network. The goal of a drop is to filter out messages, either based on the value of the tuple or in a randomized fashion, in order to rectify the overload situation and provide better overall end-to-end latency at the expense of reduced answer quality. We use loss-tolerance QoS (i.e., percent tuple delivery) as our quality metric. Our mechanism also supports query diagrams with multiple levels of windowed operators via a novel window drop operator [11].

In a distributed scenario, shedding load at a node reduces load at downstream nodes. Therefore, we can achieve higher quality outputs if we allow nodes in a chain to coordinate in choosing where and how much load to shed. We use a distributed load shedding algorithm which collects local statistics from nodes and pre-computes potential drop plans at compile time. We adjust these pre-computed plans at run-time and instantiate them through the local load shedder modules as input rates change and nodes get overloaded at varying levels. This two-phase approach provides a low-overhead mechanism for shedding load at run time, when the system is most in need of CPU cycles for running the queries.

3.3 High Availability

Stream processing applications often have weaker notions of correctness than the oft-demanded "perfect" recovery in traditional data-processing, and thus can work with weaker recovery guarantees. In Borealis, we distinguish and study three important types of failure recovery. *Gap recovery* may lose tuples and state when a failure occurs. *Rollback recovery* re-starts query-processing from a checkpoint. It does not lose any tuples, but it may produce redundant tuples. *Precise recovery*, takes over exactly from the point of failure, neither losing nor duplicating tuples.

In addition to *amnesia*, a lightweight scheme that provides high availability without incurring runtime overhead but may lose tuples during a failure, we also adapt the standard passive standby and active standby approaches to the stream processing context. In both techniques, each primary node periodically sends checkpoint messages that summarize the node state to a secondary, which takes over from the latest checkpoint when the primary fails. In addition, the upstream backup technique, where upstream nodes in the processing flow act as backups for their downstream neighbors, reduces the runtime operation overhead while trading off a small fraction of the recovery speed (and recovery guarantees). Details of our high availability approaches can be found in [8].

3.4 Parallel Processing and Dissemination

Borealis incorporates a partitioning mechanism to support operator execution in a distributed fashion. Partitioning enables the parallelization of costly operators, and a finer operator granularity to better utilize the available resources. Borealis incorporates tuple routing mechanisms, supporting data dissemination through *publication points* which match tuples in their input stream against predicates provided by subscribers to the point.

4. DEMONSTRATION DETAILS

The demonstration will illustrate various aspects of distributed operation in Borealis using a multi-player network game as the motivating application. Specifically, Borealis provides two core functionalities as the game server:

- Borealis will process continuous queries on the game state, providing aggregated views of the game world.
- Borealis will gracefully adapt to changes in resource utilization and availability.

4.1 Application: Multi-player Network Game

Our demonstration uses Borealis as the server component of the *Cube* engine [4]. Cube is an open-source first person shooter. In the game, each player represents a combatant on a three-dimensional map. Players pursue one another shooting projectiles from their inventory of weapons. The map also contains items, such as ammunition and armour. Game clients maintain and communicate these stateful map entities through the game server. For example, a player's state consists of a location, orientation and velocity.

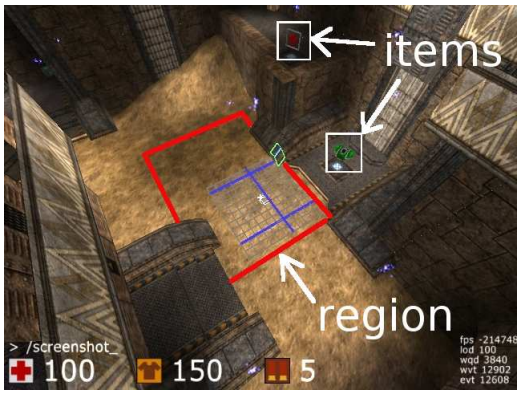

Figure 2: Cube screenshot, showing a region.

In professional gaming competitions, players participate in team competitions. Team gameplay involves a commander, or manager role, who guides individual players based on aggregated information from all team players. We extend the Cube game to support a commander role. The commander dynamically registers continuous queries to monitor the game state and implement a game strategy based on the results she gets. Hence, our queries will serve as distributed triggers that are continually evaluated to track events of interest and notify the commander in her viewport.

In this demonstration, we interject Borealis at the network layer of the Cube engine. Cube's network messages are Borealis tuples and Borealis implements the game server application logic. We run the game on a custom map, driving inputs from our workload generator. The map consists of several buildings, each of which represents a team's base. We consider two types of primitives over which continuous queries may be specified: *regions* and *entities* (i.e., players and items) in the game world. The commander view supports the selection of regions (as seen in Figure 2) and entities to query dynamically. Some examples of continuous queries are:

- Detect when more than 20 enemy combatants are near my team's base (expressed as a join over regions).
- Detect when total health of the team falls below 50% (expressed as an aggregate on players and a filter).

4.2 System and Performance Visualization

The Borealis system's monitoring tools will display in real time the current configuration (including information on which machines are executing particular boxes) and the performance of the run-time system. Our performance monitoring tool will primarily be used to visualize end-to-end tuple latency, and tuple loss through the use of sequence numbers.

Our demonstration includes a team commander's view side-by-side with the performance visualizer. The commander view provides a three-dimensional view of the game world, from a top-down vantage point. Our continuous queries' outputs act as notifications to the commander, represented in either a textual form (e.g., for counters), or a visual form (e.g., coloring players).

4.3 Setup

In our demonstration, Borealis runs on a local-area network of laptop machines. The continuous queries for both the game and commander run in the background and are automatically distributed across the machines running Borealis servers. Additionally, one machine acts as a workload generator that will introduce adjustable load into the system both in terms of the number of scripted players in the game world, and the frequency of updates from these players.

Our demonstration highlights the effect of the aforementioned Borealis features to gracefully handle an ever-increasing workload. We commence with a single Borealis server running the game query. As the game load increases, we introduce an additional Borealis server, and execute our load distribution algorithm to balance the load across the machines by moving operators. Consequently, tuple latencies as seen in the performance visualizer, will be kept low and the game experience as witnessed in the commander view, satisfactory.

At this stage, our workload generator continues to ramp up the number of players and their update frequencies. Our two Borealis servers available will be in an overloaded state, triggering our distributed load shedding algorithm. Subsequently, end-to-end latencies drop, yet we witness the loss of tuples in our performance visualizer. Players in the commander view will move less smoothly, and jump from spot to spot in the game world.

To demonstrate high availability, we will disconnect one of the machines from the network. The high availability mechanism will automatically react to this failure by having another machine take over the processing of the "crashed" machine. The participants will probably observe a "blip" in the latency immediately after the failure, during the recovery period, but the game will continue to operate.

5. ACKNOWLEDGEMENTS

We thank all members of the Borealis project for their valuable comments and continual support.

6. REFERENCES

[1] A. Arasu et. al. STREAM: The Stanford Stream Data Manager. In *ACM SIGMOD Conference*, June 2003.

[2] A. Arasu et. al. Linear Road: A Stream Data Management Benchmark. In *VLDB Conference*, August 2004.

[3] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL Workshop*, September 2003.

[4] Cube. http://www.cubeengine.com.

[5] D. Abadi et. al. Aurora: A Data Stream Management System. In *ACM SIGMOD Conference*, June 2003.

[6] D. Abadi et. al. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, January 2005.

[7] D. Carney et. al. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, September 2003.

[8] J-H. Hwang et. al. High-Availability Algorithms for Distributed Stream Processing. In *IEEE ICDE Conference*, April 2005.

[9] N. Tatbul et. al. Load Shedding in a Data Stream Manager. In *VLDB Conference*, September 2003.

[10] S. Chandrasekaran et. al. TelegraphCQ: Continuous Dataflow Processing. In *ACM SIGMOD Conference*, June 2003.

[11] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Data Streams. Technical Report CS-04-13, Brown University, Computer Science, November 2004.

[12] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *IEEE ICDE Conference*, April 2005.

[13] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26(1), March 2003.