The Medusa Distributed Stream-Processing System *

Magdalena Balazinska, Hari Balakrishnan, Jon Salz, and Mike Stonebraker,
MIT Computer Science and Artificial Intelligence Lab
http://nms.lcs.mit.edu/projects/medusa/

1 Introduction

There is a large class of emerging applications in which data, generated in a distributed environment, is pushed asynchronously to servers for processing. Some example applications for which this "push" model for data processing is appropriate include financial services (*e.g.*, price feeds), asset-tracking services (*e.g.*, reporting the status of objects and equipment in real-time), fabrication line management (*e.g.*, real-time monitoring and control of manufacturing systems), network management (*e.g.*, intrusion detection), medical applications (*e.g.*, monitoring devices and sensors attached to patients), environmental sensor/actuator systems (*e.g.*, climate, traffic, building, bridge monitoring), and military applications (*e.g.*, missile or target detection).

Several characteristics distinguish stream-processing applications from more classical data processing applications. First, unlike a traditional database management system (DBMS) where queries made by "active" human users operate on "passive" stored data, stream processing requires "active" streaming data to be processed by "passive" *continuous queries* that run for long periods of time. Second, stream processing applications require large volumes of data to be processed, with rates varying with time and often exceeding tens of thousands of messages a second. It is possible to model the bulk of the processing required in these applications in terms of standard well-defined streaming operators, such as filters, windowed aggregates, windowed joins, etc. Third, stream processing applications are naturally distributed. Many applications including weather monitoring, traffic management, and financial feed analysis process data from either geographically distributed sources or different autonomous organizations.

Early efforts in stream-oriented processing have focused on designing new operators and new languages [1, 15], as well as building high-performance engines operating at a single site [2, 5, 9]. More recently, the attention has shifted toward extending these engines to distributed environments [6, 7].

Medusa is a distributed stream-processing system built using Aurora [1] as the single-site processing engine. Medusa takes Aurora queries and distributes them across multiple nodes. These nodes can all be under the control of one entity or can be organized as a loosely coupled federation under the control of different autonomous participants.

A distributed stream-processing system such as Medusa offers several benefits:

- It allows stream processing to be incrementally scaled over multiple nodes.
- It enables high-availability because the processing nodes can monitor and take over for each other when failures occur.
- It allows the composition of stream feeds from different participants to produce end-to-end services, and to take advantage from the distribution inherent in many stream processing applications (e.g., climate monitoring, financial analysis, etc.).
- It allows participants to cope with load spikes without individually having to maintain and administer the computing, network, and storage resources required for peak operation. When organized as a loosely coupled federated system, load movements between participants based on pre-defined contracts can significantly improve performance.

Medusa facilitates query distribution and composition. It also provides schemes for high-availability and load management. In the following sections we describe the Medusa system architecture. We also present an overview of high-availability and load management.

2 Stream Processing

In stream-processing applications, *data streams* produced by sensors or other data sources are composed and aggregated by *operators* to produce some output of interest. A

*This material is based upon work supported by the National Science Foundation under Grant No. 0205445 Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

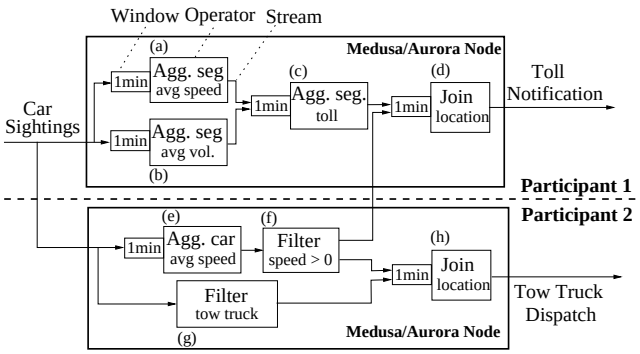


Figure 1. Example of distributed Medusa query.

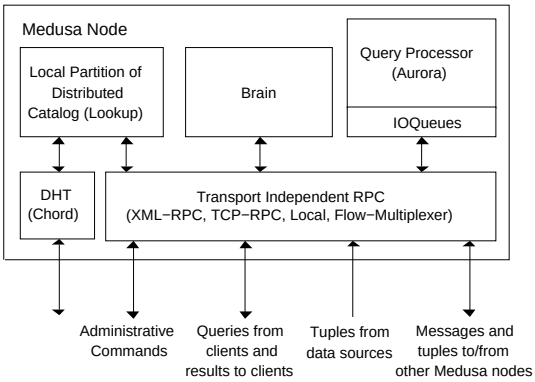


Figure 2. Medusa software structure.

data stream is a continuous sequence of attribute-value tuples that all conform to some pre-defined schema (sequence of typed attributes). Operators are functions that transform one or more input streams into one or more output streams. A loop-free, directed graph of operators is called a *query network* and all queries are *continuous*, because they continuously processes tuples pushed on their input streams.

Figure 1 shows an example of Medusa/Aurora query using a subset of the Aurora operators. The query takes a stream of "car sightings" as input, and produces streams of "toll notifications" and "tow truck dispatch". The query first applies two *windowed aggregate* operators to compute the average speed (a) and traffic volume (b) on each segment of road, every minute. These values are then used to compute tolls on these segments (c). Toll values are in turn *joined* (d) with car locations to produce toll notifications to these cars. Only cars whose speed is greater than zero (e and f) are billed. The query also *filters* (g) cars identified as tow trucks and *joins* (h) them, on the location field, with cars that have broken down.

The phrases in italics in the previous paragraph correspond to well-defined operators. While the system allows user-defined code to be written in its runtime, our expectation and experience with a few applications suggests that most applications will implement large amounts of their logic with the built-in operators. In addition to simplifying application construction and providing query optimization opportunities, using the in-built operators facilitates Medusa's task movements.

3 System Architecture

Medusa is a distributed infrastructure that provides service delivery among a collection of *participants*. The infrastructure is designed as an application-level overlay network. A Medusa participant is a financial and administrative entity that is capable of entering into Medusa *contracts*. Each participant owns and administers a collections of overlay-network nodes, sensors, and sensor proxies. A participant may or may not provide query resolution capabilities. There is a single global name space for participants, and each participant has a unique global name.

3.1 Software Components

Figure 2 shows the software structure of a Medusa node. There are two components in addition to the Aurora query processor: `Lookup` and `Brain`.

The `Lookup` component is a client of an inter-node catalog that holds information on streams, schemas, and queries running in the system. Each participant manages and administers its own catalog. The catalog allows Medusa nodes to seamless communicate each other information on the objects that exist in the system and the objects' current locations. Such information allows all nodes to carry out client requests that operate on any of these objects. For instance, Medusa clients push events on streams by sending them to any node. Nodes lookup the detailed stream information and forward tuples to appropriate locations. Similarly, when a client submits a query, it only needs to name the streams on which the query operates. Medusa nodes seamlessly bind queries to their input streams and setup all necessary tuple forwarding. The catalog can be implemented as a central server or it can be distributed using, for instance a distributed hash-table such as Chord [19]. The latter approach avoids the single point of failure and splits the burden of servicing the catalog across all Medusa nodes within a participant.

The `Brain` component handles definitions of new schemas or streams and handles query setup operations. `Brain` components at different nodes communicate with each other to re-allocate queries and improve load distribution. To do so, each `Brain` monitors local load using information about the queues (`IOQueues`) feeding Aurora and connecting query processors on separate nodes. It also uses statistics on individual box load provided by Aurora. The `Brain` uses this information to take selfish load manage-

ment decisions that together converge to good load distribution. `Brain` also handles failure recovery. Nodes detect failures of other nodes through the incapacity of their `IOQueues` to forward tuples or by periodically sending keepalive messages to subsets of other nodes. When a node detects a failure, it informs a pre-assigned secondary, which takes over all queries and tuple forwarding that were previously under the responsibility of the failed node.

To move operators with a relatively low effort and overhead compared to full-blown process migration, Medusa participants use *remote definitions*. A remote definition maps an operator defined at a node on to an operator defined at another. At runtime, when a path of operators in the boxes-and-arrows diagram needs to be moved to another node, all that's required is for the corresponding operators to be instantiated remotely and for the incoming streams to be diverted to the appropriately named inputs on the new node.

For some operators, internal operator state may need to be moved when a task moves between machines, unless some "amnesia" is acceptable to the application. Our current prototype restarts operator processing after a move from a fresh state and the most recent position of the input streams. To minimize the amount of state moved, we are exploring freezing operators around the windows of tuples on which they operate, rather than at random instants. When Medusa moves an operator or a group of operators, it handles the forwarding of tuples to their new locations.

3.2 System API

Table 1 summarizes the API through which clients communicate with Medusa. Medusa implements this API as RPC. To support a variety of clients and facilitate application development, Medusa handles both TCP-RPC and XML-RPC.

Within the Medusa library, a `MedusaClient` class facilitates application development even further by wrapping RPC calls inside simple function calls and providing some utilities such as a `start_query` method that changes the query status to `RUNNING` and an event loop. Appendix 6 shows code snippets from a typical Medusa client application.

4 Load Management

Medusa employs an agoric system model to create incentives for autonomous participants to handle each others' load. Clients outside the system pay Medusa participants for processing their queries and Medusa participants pay each other to handle load.

Unlike other computational economies that implement global markets to set resource prices at runtime, our mechanism is based on *pairwise contracts* negotiated offline between participants. These contracts set tightly *bounded prices* for migrating each unit of load between two participants and specify the set of tasks that each is willing to execute on behalf of the other. Compared to previous approaches, participants therefore have tight control over their interactions with others—they decide with whom to establish contracts, they can constrain the price of each unit of load transferred even to a fixed price (thus controlling the predictability and variability of prices), and they can constrain the tasks they will shed or accept. The mechanism also has a lower runtime overhead and efficiently re-allocates excess load without excessive load migrations even under changing load conditions. Such tight control and reduced load movements are desirable in federations where participants are real economic entities whose goal is to operate within their capacity and not necessarily achieve optimal load balance.

`create_schema`: Defines a new schema with a unique name.
`create_stream`: Defines a new stream with a unique name and associated with a previously defined schema.
`create_query`: Defines a new query that operates on pre-defined streams. `create_query_xml` is a convenience method through which clients can submit an xml query description directly.
`set_query_status`: Changes the status of a pre-defined query. Possible stata are `SETUP`, `RUNNING`, `STOPPED`, and `DELETED`. `RUNNING`
`subscribe`: Subscribes a client to receive all tuples on a named stream. The client must then listen for tuples.
`receive_events`: Pulls Medusa for tuples on a specific stream.
`post_event`: Pushes tuples to Medusa on a named stream.
`lookup_object`: Looks-up the description of an object in a participant's catalog.

Table 1. Medusa system API.

With this *bounded-price mechanism*, runtime load transfers between participants seeking to shed load and those willing to accept load occur only between participants that have pre-negotiated contracts, and at a unit price within the contracted range. The load transfer mechanism is simple and easy to implement: a participant moves load to another if the cost of processing a task locally is larger than the payment it would have to make to the other participant for the processing (plus the migration cost). The bounded-price mechanism provides incentives for selfish participants to handle each other's excess load, improving the system's load distribution. It is sufficient for contracts to specify a small price-range, for the mechanism to produce acceptable allocations where either *no* participant operates above its capacity or if the system as a whole is overloaded, then *all* participants operate above their capacity.

Although motivated by stream processing, our load management mechanism is applicable to a variety of federated systems, such as peer-to-peer systems [8, 10, 14, 16, 19, 22, 23], Web services, and cross-company workflows where

the end-to-end service requires processing by different organizations [3, 13], computational grids [4, 11, 18, 21], and overlay-based computing platforms such as Planetlab [17].

Figure 3 shows the simulation results of a 995-node Medusa system running the bounded-price load management mechanism. Figure 3(a) shows that convergence from an unbalanced load assignment to an almost optimal distribution is fast with our approach. Figure 3(b) shows the excess load remaining at various nodes for increasing numbers of contracts. A minimum of just seven contracts per node in a network of 995 nodes ensures that all nodes operate within capacity when capacity exists in the system. The key advantages of our approach over previous distributed load management schemes are (1) incentives for selfish participants to collaborate, (2) lower runtime overhead, (3) fast convergence to acceptable allocations, and (4) relatively invariant prices that a participant pays another for processing a unit of load.

5 High Availability

We are also currently exploring the runtime overhead and recovery time tradeoffs between different approaches to achieve high-availability (HA) in distributed stream processing. These approaches range from classical Tandem-style process-pairs[tandem] to using upstream nodes in the processing flow as backup for their downstream neighbors. Different approaches also provide different recovery semantics where either: (1) some tuples are lost, (2) some tuples are re-processed, or (3) operations take-over precisely where the failure happened. We discuss these algorithms in more detail in [12]. An important HA goal for the future is handling network partitions in addition to individual node failures.

6 Conclusion

Medusa is an ongoing effort. Our current prototype supports distributed operation. It provides load management with the bounded-price algorithm and simple high-availability. Our current efforts focus on moving operators' state and adding support for handling network partitions.

We have developed several applications for Medusa. One application provides indoor object tracking using RFID readers and antennas [20]. Users can track their personal belongings through a graphical interface or they can receive notifications on intelligent displays or on their cell phones. As a second application, we have integrated our project with the sensor-network project MIST []. We transformed a sensor-network access point into a Medusa client providing fire alerts and notifications.

Acknowledgments

We thank Hiroyoshi Iwashima for his contributions to early versions of Medusa. We also thank K. Lee Tang for his extensive help with the RFID-based object-tracking application.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, Sept. 2003.

[2] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[3] P. Bhoj, S. Singhal, and S. Chutani. SLA management in federated environments. Technical Report HPL-98-203, Hewlett-Packard Company, 1998.

[4] R. Buuya, H. Stockinger, J. Giddy, and D. Abramson. Economic models for management of resources in peer-to-peer and grid computing. In *Proc. of SPIE International Symposium on The Convergence of Information Technologies and Communications (ITCom 2001)*, Aug. 2001.

[5] D. Carney, U. Çetintemel, 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 *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Aug. 2002.

[6] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[8] B. Chun, Y. Fu, and A. Vahdat. Bootstrapping a distributed computational economy with peer-to-peer bartering. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[9] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

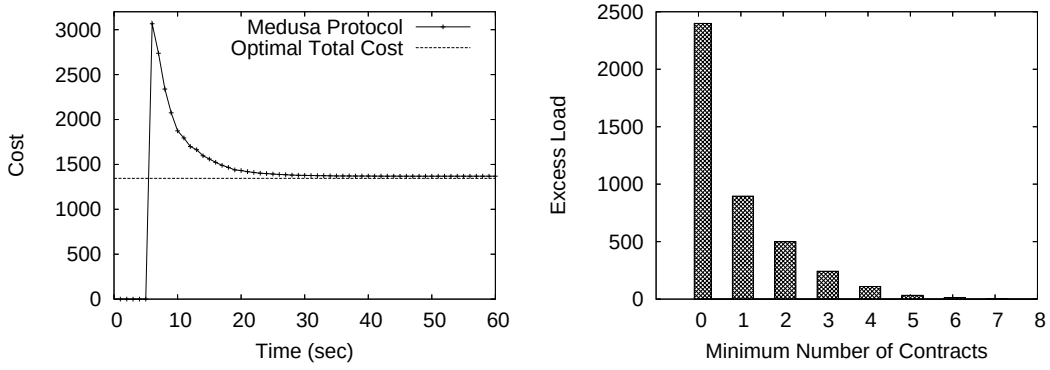
[10] R. Dingledine, M. J. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[11] I. T. Foster and C. Kesselman. Computational grids. In *Proc. of the Vector and Parallel Processing (VECPAR)*, June 2001.

[12] J.-H. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. A comparison of stream-oriented high-availability algorithms. Technical Report CS-03-17, Department of Computer Science, Brown University, Oct. 2003.

[13] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for Web services. Technical Report RC22456, IBM Corporation, May 2002.

[14] K. Lai, M. Feldman, I. Stoica, and J. Chuang. Incentives for cooperation in peer-to-peer networks. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.



(a) Convergence speed with a minimum of seven contracts per node.

(b) Final allocation for increasing numbers of contracts.

Figure 3. Performance of Medusa load management protocol

[15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[16] T.-W. J. Ngan, D. S. Wallach, and P. Druschel. Enforcing fair sharing of peer-to-peer resources. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[17] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the First Workshop on Hot Topics in Networks*, Oct. 2002.

[18] F. Schintke, T. Schtt, and A. Reinefeld. A framework for self-optimizing grids using P2P components. In *14th Intl. Workshop on Database and Expert Systems Applications (DEXA'03)*, Sept. 2003.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM Conference*, pages 149–160, Aug. 2001.

[20] K. L. Tang, M. Balazinska, J. Salz, and H. Balakrishnan. RFID-track Project. `http://nms.lcs.mit.edu/projects/slam/rfidtrack/`.

[21] The Condor Project. Condor high throughput computing. `http://www.cs.wisc.edu/condor/`.

[22] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing. In *Proc. of the Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[23] B. Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar. 2002.

Appendix: Sample Medusa Application

In this section, we present a few code snippets from a typical Medusa client.

6.1 Creating a schema

The `MedusaClient` class is designed to facilitate application development. Client applications may perform function calls on a `client` rather than directly handling the `RPC` calls themselves.

To talk to Medusa, a client must know the IP and port information for at least one node (`medusa_ip` and `medusa_port`).

```
MedusaClient client(InetAddress(medusa_ip,medusa_port));
string schema_xml = string() +
    "<schema name=\"medusa://nms.lcs.mit.edu/locator/person\">\n" +
    " <field name=\"person_name\" type=\"string\" size=\"32\"/>\n" +
    " <field name=\"person_id\" type=\"int\"/>\n" +
    "</schema>\n";
Schema schema;
Status s = schema.from_xml(schema_xml);
if ( !s )
    FATAL << ‘‘Failed reading schema from xml ‘‘ << s;

RPC<void> r = client.create_schema(schema);
if (!r.stat())
    FATAL << ‘‘Failed creating schema ‘‘ << r;
```

6.2 Creating a Schema

```
StreamDef stream_def("medusa://nms.lcs.mit.edu/locator/people", schema);
RPC<void> r = client.create_stream(stream_def);
if (!r.stat())
    FATAL << ‘‘Failed creating stream’’ << r;
```

6.3 Submitting a Query

```
string query_xml = string() +
    "<query name=\"medusa://nms.lcs.mit.edu/locator/FindMrX\">\n" +
    " <box name=\"foo\" type=\"filter\">\n" +
    "  <input port=\"1\" stream=\"medusa://nms.lcs.mit.edu/locator/people\"/>\n" +
    "  <output port=\"1\" stream=\"medusa://nms.lcs.mit.edu/locator/MrX\"/>\n" +
    "  <param name=\"nexpression.0\" value=\"person_id = 2\"/>\n"  +
    "  <param name=\"pass-on-false-port\" value=\"0\"/>\n" +
    " </box>\n"
    "</query>\n";

RPC<void> r = client.create_query_xml(query_xml);
if (!r.stat())
    FATAL << ‘‘Failed creating query ‘‘ << r;
```

6.4 Subscribing to a Stream

```
// Setting up binding to listen for tuples
RPCBindings bindings;
TCPRPCAcceptor acceptor(client.loop(), InetAddress(my_ip,my_port), bindings);

EventSink my_tcp_sink;
bindings.bind(my_tcp_sink, "MyTCPSink");
```

```
// print_event will be called every time a set of tuples appears on the stream
my_tcp_sink.on_event(wrap(&print_event));

RPC<void> r = client.subscribe(Subscription("medusa://nms.lcs.mit.edu/locator/MrX",
                               "tcp",to_string(acceptor.get_socket().getsockname()),
                               "MyTCPSink"),
                               Subscription::ADD);
if (!r.stat())
    FATAL << "Failed to add subscription: " << r;
```

6.5 Starting Query

```
// start_query calls set_query_status(RUNNING)
r = client.start_query("medusa://nms.lcs.mit.edu/locator/FindMrX");
if (!r.stat())
    FATAL << ‘‘Failed starting query ‘‘ << r;
```