High-performance Event Stream Processing Infrastructures and Applications with System S

Buğra Gedik
IBM T. J. Watson Research
Data-Intensive Systems and Analytics
Outline

1. Stream Computing
2. System S Platform
3. Distributed Middleware
4. Programming Model
5. Research Directions
1. Stream Computing
Stream Computing Paradigm

- Ingest readings from collections of software or hardware sensors in *stream* form
  - Stream: an infinite series of tuples
  - A tuple usually corresponds to an event

- Analyze the data, and produce actionable results, possibly in stream form as well

- Move away from the “store and then process” model, toward “on-the-fly processing” of real-time data
  - Similar to CQs from databases, but more general
  - Similar in spirit to Unix pipelines, but more structured

- Application areas are very broad
  - financial services, fraud and anomaly detection, manufacturing, health monitoring, environmental monitoring and surveillance, scientific computing such as radio-astronomy, etc.
Characteristics of Stream Processing Applications

- Streaming data
  - Data decomposable into meaningful units
  - Data has timeliness
    - Arrival times – not completely sporadic
    - Time to live – cannot store and process
- Streaming analysis
  - Continuous and long-running requirement
  - Time-to-Respond requirement
- Decomposable processing
  - Processing decomposable into distributed functional units
    - Can exploit parallelism and distributed resource infrastructure
- Dynamics
  - Source, data, and resource variability
  - Analytic variability
- Scale and Performance
  - Requires higher performance than store-and-process
Applications and Data Sources

- **What are the sources?**
  - Where does data come from?
  - Streaming Sources, Stored Data?
  - Is it a pull or a push from source?

- **Application and data source examples**
  - Dynamic test sequencing to minimize time-to-detect fail chips
    - Automated chip testers + stored history of test results
      - Streaming data, multiple sources – pushed to application
      - Stored data – pulled from data repository
  - Automatic trading
    - Market data as OPRA messages
      - Pushed in via UDP multicast by SIAC
  - Early prediction of infections in premature babies
    - Readings from multiple physiological sensors + Context
      - Streaming data, single data hub – pushed from physiological sensor hub
  - Extracting social network edges and nodes from call detail records
    - CDRs from network switches
      - Streaming data, pushed from switch (sometimes pull from file source)
  - Detecting anomalous patterns in intrusion detection alerts
    - Intrusion detection devices
      - Streaming data, multiple sources – pushed to application
Structure of Stream Processing Applications

- Operators, interconnected by streams
- Operators implement algorithms for data analysis, such as parsing, filtering, feature extraction, classification.

- Data-flow graphs
  - Logical: Operators, ports, streams
  - Physical: Processing Elements, Hosts
Logical View vs Physical View of a SP Application

OP Graph <-> PE Graph

Operators -> Processing Elements -> Hosts
2. System S Platform
System S

- System S is a distributed stream processing platform under development at IBM
  - Started as a research project at the IBM Thomas J. Watson Research Center in 2003
  - Multi-disciplinary effort: High Performance Systems, Programming Languages, Knowledge Representation, Data Management, to Optimization, and Analytics

- Productization under the name *InfoSphere Streams*

- System S technology enables
  - construction of applications that can respond quickly to events and changing requirements, adapt rapidly to changing workloads, and continuously analyze real-time data at high rates

- System S targets cluster of machines
  - 100s of hosts, homogeneous architecture (with some experience in heterogeneous environments)
System S Computing Platform

- **A Distributed Runtime**: System S runtime provides an execution substrate for streaming applications
  - advanced job management
  - resource allocation and scheduling
  - high performance data transport
  - security
  - high availability

- **Language and Library Support**: System S applications are developed using the Spade language
  - Flow composition
  - Flow manipulation
  - Extensibility
  - Data Sharing

- **Integrated Development Environment**: System S provides an eclipse-based IDE for developing streaming applications using the Spade language. The development environment also includes support for interacting with the System S runtime capabilities and *visualization* of running jobs.

- **Configuration and Administration**: System S includes web-based interfaces as well as command line tooling for configuring and administering System S instances in *multi-user* environments (with *security*).
Application Development Paradigm

Source Adapters

Operator Repository

Sink Adapters

**SPADE**: Stream processing dataflow language

Platform Optimized Compilation
Application Development Paradigm

- Reusable set of operators
- Connectors to external static or streaming data sources and sinks

SPADE: Stream processing dataflow language

Platform Optimized Compilation
Runtime Services

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters.
Runtime Services

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation.

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters.
Runtime Services

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation.

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters.
Runtime Services

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation.

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters.
Runtime Services

Adapts to changes in resources, workload, data rates

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters
Runtime Services

Adapts to changes in resources, workload, data rates

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters

Capable of exploiting specialized hardware

System S Data Fabric

Operating System

Transport

X86 Blade

FPGA Blade

Cell Blade

X86 Blade

X86 Blade
Runtime Services

Adapts to changes in resources, workload, data rates

Optimizing scheduler assigns operators to processing nodes, and continually manages resource allocation

Runs on commodity hardware – from single node to blade centers to high performance multi-rack clusters

Capable of exploiting specialized hardware

Processing Element Container
Processing Element Container
Processing Element Container
Processing Element Container
Processing Element Container

System S Data Fabric
Transport
Operating System

BG Node
BG node
BG node
BG node
BG node
Job Management

- Job: A flow graph of PEs
- Basic job management
  - Submit/cancel jobs
  - Restart/move jobs
  - Monitor status of jobs
- Advanced job management
  - Evolving applications: dynamic set of jobs
  - Inter-job connections: imported/exported streams
    - static connections
    - dynamic connections
  - Launching applications from within applications
- Logical versus physical state
  - Logical state
    - Job/PE status, connection info, import/export info, etc.
  - Physical state
    - Host liveness, PE liveness, resource usage (host/pes), performance metrics (system/application)
Resource Monitoring

- Scalability/Performance <-> Rate of communication
  - Streams: data path, high rate of communication (custom transport)
  - Job management: control path, low rate of communication (CORBA calls)
  - Resource monitoring: still control, but the rate of communication may not be low (TMON protocol)

- Resource monitoring and TMON
  - Queries on resource usage, or performance metrics (SCH <-> SRM)
  - Each host controller has a *probe*, that collect metrics
  - The host controller also participates in the formation of metric collection *tree*
  - The tree is a distributed multi-parent (for resiliency) tree routed at SRM
  - Scalable data collection
    - minimizes the number of connections per host
    - performs data reduction for aggregate queries
    - performs data compression for non-aggregate queries
Scheduling & Placement

- **Optimization knobs**
  - Operators to PEs: intra-process connections (function calls), versus inter-process connections (going through transport)
  - Threads to Operators: pipelined parallelism, data parallelism
  - PEs to hosts: load balancing, admission control

- **Optimization times**
  - Compile-time: efficient if workload and resource characteristics are static
  - Run-time: more adaptable, but has some overhead (observe + decide + act)

- **Profile-driven optimization**
  - Resource functions
    - How the operators react to changes in rate
    - Not necessarily linear (e.g.: time based join)
  - Continuous learning during deployment
    - Good for run-time optimizations
    - Difficult to exercise different operating points
  - Offline profiling pre-deployment
    - Good for compile-time optimizations, more control
    - Cannot adapt to changing environment

- **Interaction between compile-time and run-time optimizations**
Compile-time Optimizations

- Operator fusion
  - Operators to PEs, streams to function calls
  - Fuse all: won’t fit a single node in large apps
  - PE per operator: too much intra-process communication

- How to come up with a performant and compliant fusion scheme
  - **Performant**: Use profiling information to derive a good fusion
  - **Compliant**: Use a constraint checker to satisfy fusion constraints

- Fusion constraints (NP-hard problem)
  - PE co-location: E.g., Shared per-process resources, like memory
  - PE ex-location: E.g., Fault tolerance, memory protection
  - Host co-location: E.g., Shared per-node resources, like licenses, co-processors, etc.
  - Host ex-location: E.g., High availability, single point of failure
  - Host isolation: E.g., Performance considerations, hardware limitations (BG)

- Optimization heuristics
  - Bottom up: Start with PE per OP, fuse until *effective utilization* reaches a threshold or the PE becomes too big, whichever comes first
  - Top down: Start with a single PE, progressively divide until you reach PEs that fit
Run-time Optimizations

Current runtime system state

What if I deploy this new application?

New estimated runtime system state

Lightly loaded host

Heavily loaded host

Reasonably loaded host

Overloaded host

Heavily loaded host

Host 3
Run-time Scheduling & Optimizations

- **PE placement**
  - Decide where each PE would run
  - Under extreme conditions, *move*

- **Admission control**
  - Rank based admission control
  - What if the workload characteristics change and what used to fit does not fit any more?

- **Make use of data parallelism**
  - Within an operator
    - Elastic operators: fork, process in parallel, merge
    - Adjust the number of threads dynamically
    - Adapt to resource availability
  - Across OPs/PEs-hosts
    - Dynamically dispatch more operators

- **Pipelined parallelism**
  - Assign multiple threads to an operator sub-graph
Data Transport

- **Basics**
  - Used to transfer tuples between PEs
  - Critical for achieving low-latency and high-throughput
  - Needs to be reliable and order preserving

- **Several in-house implementations exist**
  - TCP-based transport + epoll for handling multiple connections
  - BGSM - BlueGene transport that uses RDMA

- **LLM - IBM Websphere MQ Low Latency Messaging**
  - Based on research done in IBM Haifa
  - Multi-threaded transport with custom buffering and time-outs
  - Adjust the trade-off between latency and throughput
  - Support for Infiniband switches
  - Support for reliable multi-cast over UDP
    - Critical for large fan-out scenarios

- **Facade tuples (minimum copy transfers)**
  - Keep the data in the buffer received from the transport
  - Do not marshal/unmarshal, but still provide an OO interface
High Availability

- Middleware-level (control path)
  - Restart & log to transactional store
    - Restart daemons (potentially on a different node) upon failure
    - Non-idempotent RPCs are transactional and backed up by a DB
  - Replication
    - Name server entries are replicated over multiple nodes

- Application-level (data level)
  - Checkpointing & PE restart
    - When tuple loss is not critical
  - Replicated segments
    - When tuple loss is critical
  - Partial fault tolerance
    - Language controlled

- Operator-level
  - Incremental checkpointing of join windows
  - Customized checkpointing based on operator semantics
3. Programming Model
The SPADE Language

➡ What is SPADE?
   – A rapid application development front-end for System S

➡ What does it consist of?
   – A language, a compiler, and auxiliary support for building distributed stream-processing applications

➡ What does it provide?
1. A language to compose parameterizable distributed stream processing applications, in the form of data flow graphs
   • Basic concepts: Operators, ports, streams
   • Custom operators in C++/(Java) (no restrictions)
   • Node pools, placement, fusion, etc.
2. A type-generic streaming operator model
   • Basic concepts: windows on input streams, configuration parameters, output attribute assignments, aggregation functions, etc.
3. A set of built-in operators - the relational toolkit
4. Support for extending the language with additional toolkits: new type-generic, configurable, and reusable operators
The SPADE Language

- Flow composition
- Flow manipulation
- Extensibility
- Data Sharing
Flow Composition

- Distributed data flow graphs
  - Hosts, PEs, operators, ports, streams
    - Streams are typed using a nested type system

- Modular composition of graphs
  - Composite operators that contain sub-graphs within

- Incremental application composition
  - Exporting streams through properties
  - Importing streams through subscriptions
  - Programmatic interfaces for changing these
Flow Manipulation

- Type-generic, configurable operators
  - Stream relational operators
  - Edge adapters
  - Uniform syntax

- Spade expression language
  - Write complex functions in Spade
  - Write operator logic in Spade
  - Compiled into C++ for performance
Extensibility

- Toolkits that contain
  - Primitive operators, composite operators
  - Spade functions, native functions

- Unified operator model that captures
  - Built-in operators - comes with the standard toolkit
  - User-defined operators
    - Non-type generic, non-reusable, not highly configurable
  - User-defined built-in operators
    - Type-generic, reusable, highly configurable

- Library support for
  - Window handling
  - Common operator logic
Data Sharing

- Streams as the sole mechanism of sharing data between operators do not scale in certain use cases
  - Control variables, mailboxes, etc...
  - May end up having $nxm$ stream connections

- Shared variables as:
  - Spade language-level mechanism to share state
  - C++ level APIs to access lower-level functionality
  - Uses the same type system that regular streams have
  - Supports different caching and consistency models to support different use cases (read/write rations, freshness requirements, etc.)
Type System - Why does it matter?

- Strongly typed
- Nested types, lists, sets, maps
- Bounded lists, sets, maps (e.g. `map<string, list<int> [5] [6]`)
  - Fixed layout known at compile-time
  - *Code generation* used to provide a facade OO interface
Primitive Operators

- Implemented (defined) in a general purpose language
  - Based on code generation, written as codegen templates
  - Operator models define the valid syntax, and to a lesser degree semantics
  - Customized code based on the specific instantiation of the operators
    - Fast runtime performance
    - More time spent during compilation
      - incremental compilation, constant folding + runtime constants

- Instantiated within Spade

```csharp
stream<string buyer, string seller, string item>
Sale = Join(Bid; Quote) {
  window Bid : sliding, time(30);
  Quote : sliding, count(50);
  param match : Bid.item == Quote.item &&
    Bid.price >= Quote.price;
  output Sale : item = Bid.item;
}
```

Invoke operator “Join” on input streams “Bid” and “Quote” to define output stream “Sale”.
Primitive Operators - extensions

\[
\text{stream<}\text{string buyer, string seller, string item, int64 id>}
\]
\[
\begin{align*}
\text{Sale} &= \text{Join}(\text{Bid}; \text{Quote}) \{ \\
\text{logic} & \quad \text{state : mutable uint64 n = 0;} \\
\text{Bid} & \quad : n++;
\}\text{window} \\
\text{Bid} & \quad : \text{sliding, time(30)}; \\
\text{Quote} & \quad : \text{sliding, count(50)}; \\
\text{param} & \quad \text{match : Bid.item == Quote.item &&} \\
& \quad \text{Bid.price >= Quote.price;}
\}\text{output} \\
\text{Sale} & \quad : \text{item = Bid.item, id = n}; \\
\}
\]

A C-like statement language is supported with functions and basic flow control structures.

Statement executes when tuple arrives on stream Bid.

Local state, persists throughout execution.
Composite Operators

- Helps with modularity
  - Brings hierarchy into large programs
- Helps with reuse
  - Composite operators can be defined in spade files that are shared across applications
Configuring Composite Operators

Hygienic macro expansion
- attributes
- expressions
- operators
Configuring Composite Operators

composite SequencingFilter (output O; input I) {
  param
    attribute $seq; // output attribute to assign
    expression<boolean> $filter;
  type
    OutT = I, tuple<uint64 $seq>;
  graph
    stream<OutT> O = Functor(I) {
      logic
        state : { mutable uint64 cnt = 0; }
        I     : { cnt++; }
        param filter: $filter;
        output O : $seq = cnt;
    }
}

Definition

Hygienic macro expansion
- attributes
- expressions
- operators
Configuring Composite Operators

```plaintext
stream <int age, string name> NonSequenced := ...

stream <int age, string name, uint64 no> Sequenced := SequencingFilter(NonSequenced) {
  param
    filter: age < 30;
    seq   : no;
}

composite SequencingFilter (output O; input I) {
  param
    attribute $seq; // output attribute to assign
    expression<boolean> $filter;
  type
    OutT = I, tuple<uint64 $seq>;
  graph
    stream<OutT> O = Functor(I) {
      logic
        state : { mutable uint64 cnt = 0; }
        I     : { cnt++; }
      param filter: $filter;
      output O    : $seq = cnt;
    }
}
```

Hygienic macro expansion
- attributes
- expressions
- operators
Map Reduce with Composites
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Spade Composites

HDFS Reader

Partition

Temp Storage Reader

HDFS Writer

Controller

Master

Temp Storage Writer

Map

Reduce
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Data Location / Params
Map Module
Reduce Module

Spade Composites

HDFS Reader
Partition
Temp Storage Writer

Temp Storage Reader
HDFS Writer

Controller
Master

Map

Reduce
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Spade Composites

Controller

Data Location / Params

Master

HDFS Reader

Map Module

Partition

Temp Storage Writer

Map

Temp Storage Reader

Reduce Module

HDFS Writer

Reduce
Map Reduce with Composites

Original graph  Composite op. M  Expanded graph

Compilation

Spade Composites

Controller
Data Location / Params

HDFS Reader
Map Module
Partition
Temp Storage Writer

Map

Temp Storage Reader
Reduce Module
HDFS Writer

Reduce
Map Reduce with Composites

Compilation

Spade Composites

Master

Data Location / Params

Controller

HDFS Reader

Map Module

Map

Temp Storage Reader

Reduce Module

Temp Storage Writer

Reduce

Original graph

Composite op. M

Expanded graph
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Compilation

System-S Runtime

Spade Composites

Master

Map

Reduce

HDFS Reader

Map Module

Partition

Temp Storage Writer

Temp Storage Reader

Reduce Module

HDFS Writer

Controller

Data Location / Params
Map Reduce with Composites

Compilation

System-S Runtime

Spade Composites

Master

Map

Reduce

HDFS Reader

Map Module

Partition

Temp Storage Writer

Temp Storage Reader

Reduce Module

HDFS Writer
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Master
Map
Reduce

Compilation

System-S Runtime

Spade Composites

Controller
Data Location / Params

Master

HDFS Reader
Map Module
Partition
Temp Storage Writer

Map

Reduce

Temp Storage Reader
Reduce Module
HDFS Writer
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Spade Composites

Compilation

System-S Runtime

Master

Map

Reduce

Controller

Data Location / Params

HDFS Reader

Map Module

Partition

Temp Storage Writer

Temp Storage Reader

Reduce Module

HDFS Writer

© 2009 IBM Corporation
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Master
Map
Reduce

Compilation

System-S Runtime

Spade Composites

Controller
Data Location / Params

Master

HDFS Reader
Map Module
Partition
Temp Storage Writer

Map

Temp Storage Reader
Reduce Module
HDFS Writer

Reduce
Map Reduce with Composites

Master

Map

Reduce

Compilation

System-S Runtime

Spade Composites

Controller

Data Location / Params

HDFS Reader

Map Module

Partition

Temp Storage Writer

Temp Storage Reader

Reduce Module

HDFS Writer
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Master  Map  Reduce

Compilation

System-S Runtime

Spade Composites

Controller

Data Location / Params

HDFS Reader

Map Module

Partition

Temp Storage Writer

Temp Storage Reader

Reduce Module

HDFS Writer
Map Reduce with Composites

Original graph

Composite op. M

Expanded graph

Compilation

System-S Runtime

Spade Composites

Controller

Data Location / Params

Master

Map Module

Partition

Temp Storage Writer

Map

HDFS Reader

Reduce Module

Temp Storage Reader

HDFS Writer

Reduce
Map Reduce with Composites
Map Reduce with Composites
Import/Exports and Dynamic Composition

Similar to publish subscribe, but at the stream-level
- We are subscribing to streams with certain properties
- We are publishing properties on a given stream

Enables applications to
- Connect to each other
- Evolve through
  - incremental deployment
  - dynamic composition

Runtime APIs
- Change stream properties
- Change stream subscriptions
Import/Exports and Dynamic Composition

- Similar to publish subscribe, but at the stream-level
  - We are subscribing to streams with certain properties
  - We are publishing properties on a given stream

- Enables applications to
  - Connect to each other
  - Evolve through
    - incremental deployment
    - dynamic composition

- Runtime APIs
  - Change stream properties
  - Change stream subscriptions

App B

() = Export(E) {
  param howPublished : x="baz", y=[5,6,7];
}

App A

stream<int32 z> I = Import() {
  param subscription : x == "baz" && y[0] < 10;
  applicationScope : "mySession"; // optional
}
Sample SPADE Operators
- **Functor**: Add attributes, remove attributes, filter tuples, map output attributes to a function of input attributes
- **Aggregate**: Window-based aggregates, with group by
- **Split**: Splits a stream into multiple substreams
- **Join**: Window-based binary stream join
- **Sort**: Window-based approximate sorting
- **Barrier**: Synchronize multiple streams
- **Punctor**: Creates user-defined

Sample SPADE Adapters
- **File**: Read/write from/to files
- **Tcp**: Read/write from/to TCP sockets
- **Udp**: Read/write from/to UDP sockets
- **formats**: (xml*, binary, csv) / (compressed, uncompressed)
- **WFO, MQ, RSS, ODBC. DB2, SolidDB**
### Spade Standard Toolkit

**Sample SPADE Operators**
- **Functor**: Add attributes, remove attributes, filter tuples, map output attributes to a function of input attributes
- **Aggregate**: Window-based aggregates, with group by
- **Split**: Splits a stream into multiple substreams
- **Join**: Window-based binary stream join
- **Sort**: Window-based approximate sorting
- **Barrier**: Synchronize multiple streams
- **Punctor**: Creates user-defined

**Sample SPADE Adapters**
- **File**: Read/write from/to files
- **Tcp**: Read/write from/to TCP sockets
- **Udp**: Read/write from/to UDP sockets
- **formats**: (xml*, binary, csv) / (compressed, uncompressed)
- **WFO, MQ, RSS, ODBC. DB2, SolidDB**
Spade Standard Toolkit

Sample SPADE Operators
- **Functor**: Add attributes, remove attributes, filter tuples, map output attributes to a function of input attributes
- **Aggregate**: Window-based aggregates, with group by
- **Split**: Splits a stream into multiple substreams
- **Join**: Window-based binary stream join
- **Sort**: Window-based approximate sorting
- **Barrier**: Synchronize multiple streams
- **Punctor**: Creates user-defined

Sample SPADE Adapters
- **File**: Read/write from/to files
- **Tcp**: Read/write from/to TCP sockets
- **Udp**: Read/write from/to UDP sockets
- **formats**: (xml*, binary, csv) /
  (compressed, uncompressed)
- **WFO, MQ, RSS, ODBC, DB2, SolidDB**
**Spade Standard Toolkit**

Sample SPADE Operators
- **Functor**: Add attributes, remove attributes, filter tuples, map output attributes to a function of input attributes
- **Aggregate**: Window-based aggregates, with group by
- **Split**: Splits a stream into multiple substreams
- **Join**: Window-based binary stream join
- **Sort**: Window-based approximate sorting
- **Barrier**: Synchronize multiple streams
- **Punctor**: Creates user-defined

Sample SPADE Adapters
- **File**: Read/write from/to files
- **Tcp**: Read/write from/to TCP sockets
- **Udp**: Read/write from/to UDP sockets
- **formats**: (xml*, binary, csv) / (compressed, uncompressed)
- **WFO, MQ, RSS, ODBC. DB2, SolidDB**
Sample SPADE Operators
- **Functor**: Add attributes, remove attributes, filter tuples, map output attributes to a function of input attributes
- **Aggregate**: Window-based aggregates, with group by
- **Split**: Splits a stream into multiple substreams
- **Join**: Window-based binary stream join
- **Sort**: Window-based approximate sorting
- **Barrier**: Synchronize multiple streams
- **Punctor**: Creates user-defined

Sample SPADE Adapters
- **File**: Read/write from/to files
- **Tcp**: Read/write from/to TCP sockets
- **Udp**: Read/write from/to UDP sockets
- formats: (xml*, binary, csv) /
  (compressed, uncompressed)
- **WFO, MQ, RSS, ODBC. DB2, SolidDB**
Punctuations

- Control signals in the stream, not tuples
  - Window Punctuation: E.g.: Apply an aggregation on the results of a join
  - EOF Punctuations: Flush buffers when EOF is reached
- Rules for propagation of punctuations
- Rules for maintaining punctuation semantics
Data Sharing

(a) A user may want to associate externally modifiable control variables with operators to affect the behavior of streaming application at runtime.

(b) An operator, for achieving fault-tolerance, may want to efficiently checkpoint its state to a shared object, which can then be used by a passive standby in case of a failure.

(c) A group of operators can collaborate to build a repository of interesting events which can be watched for by another set of operators.
Configurable Transport, Caching, Consistency

visibility instantiation mutability s_prefix

public static mutable map<string, int32> s_map {
    pragma
    lifetime : eternal;
    consistency : causal;
    sizeHint : 1GB;
    writesPerSecond : 5;
    readsPerSecond : 500;
}

key-value pairs passed to the underlying implementation
Configurable Transport, Caching, Consistency

visibility instantiation mutability s_prefix

public static mutable map<string, int32> s_map {
    pragma
    lifetime : eternal;
    consistency : causal;
    sizeHint : 1GB;
    writesPerSecond : 5;
    readsPerSecond : 500;
}

key-value pairs passed to the underlying implementation

Client side

Server side

Distributed Protocol

Transport

Operator Interface

Cache

Data Server

Transport

Distributed Protocol

Data Server

Client invoke()
Configurable Transport, Caching, Consistency

visibility instantiation mutability s_prefix

public static mutable map<string, int32> s_map {
  pragma
  lifetime : eternal;
  consistency : causal;
  sizeHint : 1GB;
  writesPerSecond : 5;
  readsPerSecond : 500;
}

key-value pairs passed to the underlying implementation
Configurable Transport, Caching, Consistency

visibility  instantiation  mutability  s_prefix

public static mutable map<string, int32> s_map {
  pragma
  lifetime : eternal;
  consistency : causal;
  sizeHint : 1GB;
  writesPerSecond : 5;
  readsPerSecond : 500;
}

key-value pairs passed to the underlying implementation
Monitoring and Debugging

- Semantic Debugging
  - Verify configuration, composition, stream content
  - Composition visualizer, stream debugger (sdb)
  - Standalone applications (plain executable)
- Operator Debugging (low-level)
  - Locate interesting events via sdb
  - Jump into port logic debugging via gdb
- Deployment Debugging
  - Spade language has knobs to change fusion and placement
  - Visualize the live layout of the application
- Performance Debugging
  - Built-in metrics
  - Custom (user-defined) metrics
  - Visualization support
Additional Research Areas

- **Provenance**
  - Trace back the history of a tuple
    - Operators, streams, source and intermediate tuples
    - Partial replay, reproducibility, accountability

- **Control flow language at the composition-level**
  - Loops, conditionals, variables, first class citizens at the composition level

- **Visualization**
  - Dealing with large-scale hierarchical flow graphs
    - Detecting bottlenecks, observing performance metrics
    - How to deal with the composition-level control-flow constructs
  - Declarative dashboards
  - Log visualization

- **Non-streaming models**
  - Request/response functionality in stream processing

- **Automated composition**
  - Composing flow graphs based on semantic tagging
    - Template flows, populated automatically based on semantic tag matching via ontologies

- **Alternative hardware**
  - FPGA, Cell, GPU
Questions