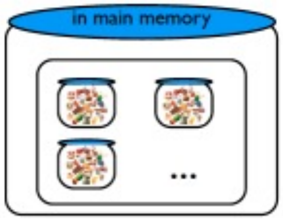
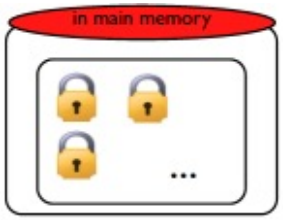
SPL Stores

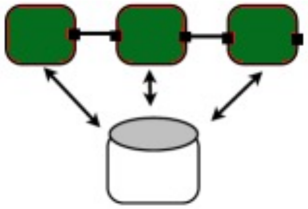


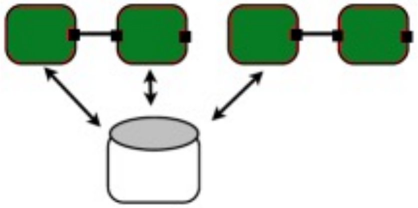


Process Store (ps)
Distributed Process Store (dps)

Senthil Nathan
[IBM T.J.Watson Research Center, New York, sen@us.ibm.com]
Buğra Gedik
[Bilkent University, Ankara, Turkey, bgedik@cs.bilkent.edu.tr]

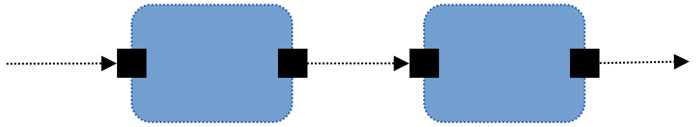
2011 - 2021



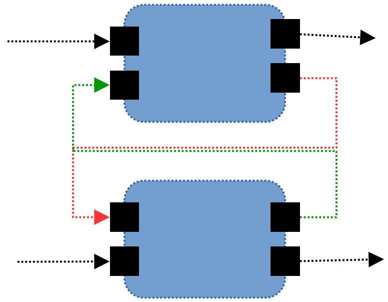


State in SPL

- All state is local to operators
  - Maintained across tuple firings
    - e.g., member variables in C++ or Java operators
    - e.g., state variables in SPL Custom operators
- Communicating state can be done via streams



  - Works well for one directional communication
    - If true sharing is needed, bidirectional streams can be used
    - This is messy, does not scale, has synchronization problems



Sharing State

- This is not an uncommon need
  - Data needs to be accessed by multiple operators running on one or more machines
- Many use cases:
  - Manage dynamic configuration data accessed by multiple operators
  - Access large-scale state that does not fit into the local memory of a single operator
  - Provide an external system access to the application data
  - Reference data look-up from multiple operators
  - And more ...

dps (distributed process store)

- dps (distributed process store) is a set of SPL native functions that enable SPL operators to share state across PEs
  - Potentially across different machines, different applications, and different Streams instances
  - Sharing of state is allowed anywhere between SPL built-in operators, SPL functions, native functions, C++, Java operators, and also with certain non-Streams applications.
- Provides a *store factory* abstraction
- Multiple stores can be created, accessed, and shared literally from anywhere within one or more Streams applications.
- Each store is a key-value map
  - Any SPL type can be used as a key and a value
  - Hybrid types are **not** supported
    - More like a traditional map<K, V> in SPL

dps storage details (1 of 2)

- dps stores key/value pairs in a back-end data store
- A back-end can be an in-memory data store or even an RDBMS as shown below.

RDBMS

DB2
Informix
MySQL
...

IBM's in-memory data store heritage

WebSphere eXtreme Scale (WXS)
-- Formerly ObjectGrid
-- Morphed a few times
-- Billy Newport's dream project

solidDB
-- Hybrid and an acquired product

HydraDB and TOAD
-- Ongoing work in Watson/Almaden

Open source in-memory data stores

1) memcached (proven and active since 2003)
2) Redis (well rounded and technically superior)

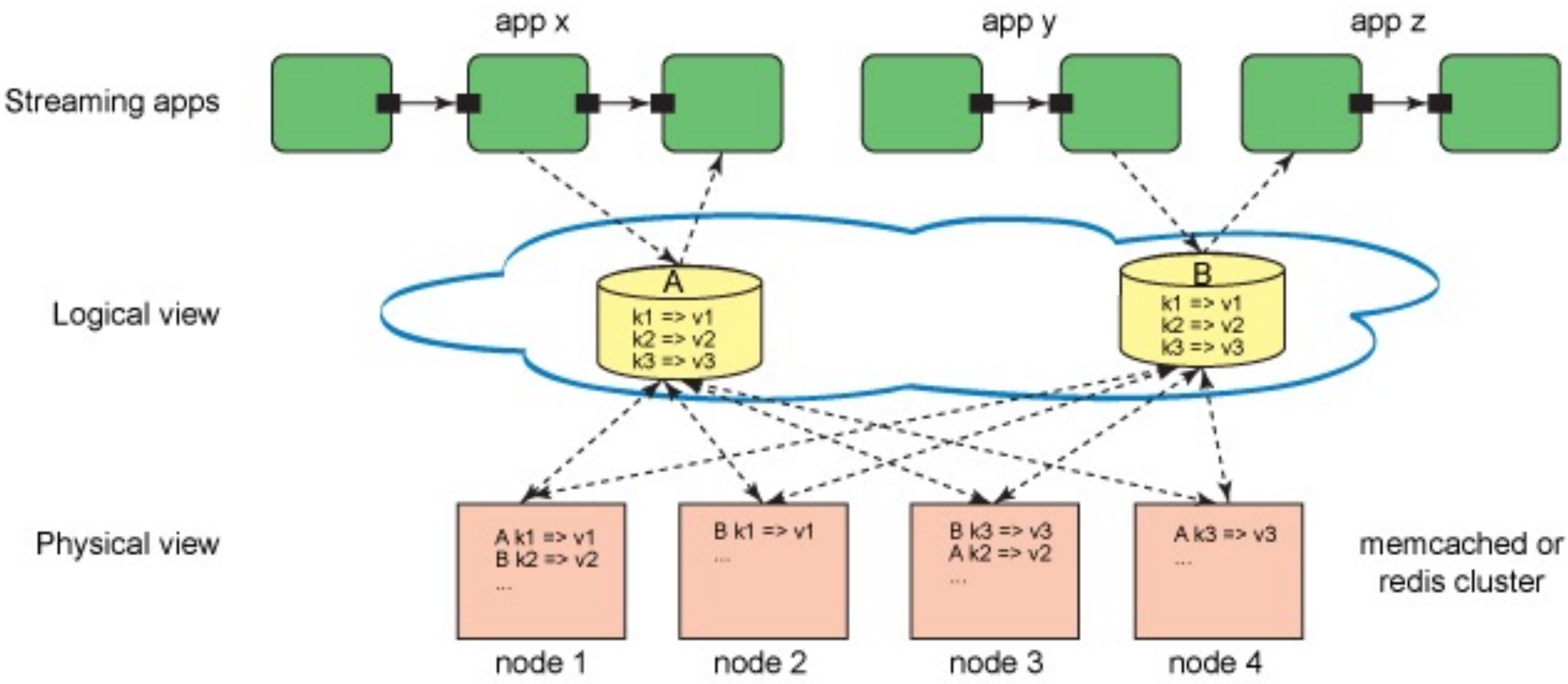
More choices galore in this crowded field:

3) Hazelcast
4) Voldemort
5) Hyperdex
6) Cassandra
7) Terracotta
8) MongoDB
9) Gemfire
10) Ehcache
11) IBM Cloudant / CouchDB
12) Aerospike
13) Couchbase
14) VoltDB
15) MarkLogic and many more

dps storage details (2 of 2)

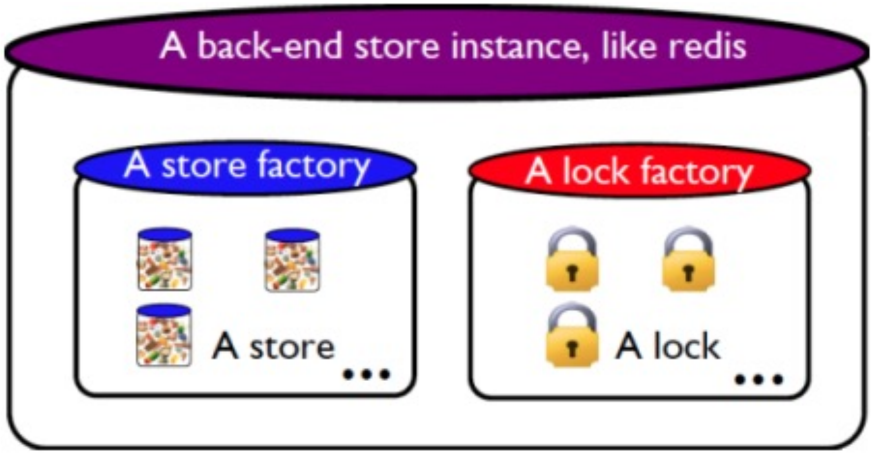
- Currently it supports nine back-end data stores:
  - A DB2 back-end exists from the work done in 2011
- Both memcached and Redis are distributed in-memory key/value stores
  - They use the memory from multiple machines to distribute the data items and achieve scale
  - Many smart firms try to make money out of them
    - http://aws.amazon.com/elasticache/
- memcached is barebones, does not provide replication (for fault tolerance), and no persistence
- Redis provides advanced fault-tolerance and persistence options

Architecture (dps)



- Unobtrusive design that lends itself to a high degree of scaling at every layer.

Inside the dps



- Inside a given dps instance:
  - A store factory allows:
    - Creation of multiple uniquely named stores
    - Removal of stores
  - A lock factory allows:
    - Creation of distributed locks
    - Acquisition and releasing of distributed locks
    - Removal of distributed locks

dps usage

- Each application puts a configuration file in its etc directory
  - Specifies the dps back-end instance configuration
- A given application can work with a single dps back-end instance at a time
- A dps instance provides a store factory and lock factory interfaces
- Multiple stores and locks can be created, and shared
- Stores and locks can be searched using their names

Supported operations (dps)

- Store factory
  - dpsCreateStore
  - dpsCreateOrGetStore
  - dpsFindStore
  - dpsRemoveStore
- Store
  - dpsGet, dpsPut, dpsRemove, dpsHas, dpsSize
  - dpsBeginIteration, dpsEndIteration, dpsNext
  - dpsSerialize, dpsDeserialize
  - … (more)

Sample code (dps)

SPL & C++

```
mutable uint64 err = 0ul;
mutable boolean res = false;
mutable uint64 s = 0ul;

rstring dummyRstring = "";
s = dpsCreateOrGetStore("Best in the language business", dummyRstring, dummyRstring, err);

res = dpsPut(s, "Fortran", "John W. Backus", err);
dpsPut(s, "C", "Dennis MacAlistair Ritchie", err);
dpsPut(s, "C++", "Bjarne Stroustrup", err);
dpsPut(s, "Java", "James Arthur Gosling", err);
dpsPut(s, "Perl", "Larry Wall", err);
dpsPut(s, "PHP", "Rasmus Lerdorf", err);
dpsPut(s, "Python", "Guido van Rossum ", err);
dpsPut(s, "Ruby", "Yukihiro Matsumoto", err);
dpsPut(s, "SPL", "Martin Hirzel, Bugra Gedik", err);

mutable uint64 size = dpsSize(s, err);
uint64 it = dpsBeginIteration(s, err);
mutable rstring key = "";
mutable rstring value = “”;

while(dpsGetNext(s, it, key, value, err)) {
   printStringLn("'" + key+"' => " + value);
}

dpsEndIteration(s, it, err);
dpsClear(s, err);
dpsRemoveStore(s, err);
```

Java Client API (dps)

- This is a sweet deal for Java developers.
- Client API is provided for access from Java operators and from outside of SPL applications

```
import com.ibm.streamsx.dps;

StoreFactory sf = DistributedStores.getStoreFactory();
Store s7 = sf.createOrGetStore(“IBMStore”, “ustring”, “int32”);
s7.put(“Eye”, 10);
s7.put(“Bee”, 11);
s7.put(“/\/\”, 12);


for (KeyValuePair kv : s7) {
   String key = kv.getKey();
   int value = (Integer)kv.getValue();

 ...
}
```

Implementation Details (dps)

- Serialization and deserialization happens when data items are put and get from the dps
- dps operations are not as cheap as the ps ones, since distributed operation is involved
- Still, the performance is reasonable:
  - e-g: Results from a bulk read/write operation using Redis:
    - 5M parallel inserts <rstring, rstring>: 14 seconds [357K puts/sec]
    - 5M parallel reads <rstring, rstring>: 12 seconds [416K gets/sec]
- memcached does not support iteration by default, so a rather involved custom implementation that relies on a segmented catalog is used
- Redis back-end has a clean straightforward implementation

Mutual exclusion with dl

- All dps operations are atomic by default
- More involved use cases bring collision and data override challenges (e-g: multiple writers, performing transactional activities etc.)
- A distributed lock (dl) library is provided to tackle that
  - dl can be used for creating mutual exclusion blocks
  - Such blocks can contain multiple store operations with an exclusive access to a store
- Lock factory and lock interfaces are provided
  - dlCreateOrGetLock, dlRemoveLock
  - dlAcquireLock, dlReleaseLock
- Recovering locks from user errors:
  - Locks can specify a lease time so that if a party fails while holding the lock, the lock is released after the lease expiry

Sample code (dl)

SPL & C++

```
mutable uint64 err = 0ul;
mutable boolean res = false;
mutable uint64 s = 0ul;
mutable uint64 l = 0ul;

rstring dummyRstring = "";
s = dpsCreateOrGetStore("Super_Duper_Store", dummyRstring, dummyRstring, err);
// Create a user defined distributed lock.
l = dlCreateOrGetLock("Super_Duper_Lock", err);
// Get a distributed lock with a lease time for 30.0 seconds.
dlAcquireLock(l, 30.0, err);

// Do a bulk store activity.
mutable int32 cnt = 0;
// Insert 1 Million rstring values in the store.
while(cnt++ < 1000000) {
   res = dpsPut(s, "myKey" + (rstring)cnt, (rstring)cnt, err);
}

// Release the lock.
dlReleaseLock(l, err);
// Remove the lock only if needed. Typically, locks will stay around until the application ends.
dlRemoveLock(l, err);
uint64 size = dpsSize(s, err);
dpsRemoveStore(s, err);
```

Error handling (dps)

- After every dps operation, error code and error message (if any) can be read using these APIs.
  - uint64 dpsGetLastStoreErrorCode()
  - rstring dpsGetLastStoreErrorString()

SPL & C++

```
uint64 s = 0ul;
uint64 err  = 0ul;
rstring dummyRstring = “”;

s = dpsCreateOrGetStore("Zip_Code_Lookup", dummyRstring, dummyRstring, err);

if (err != 0ul) {
   printStringLn("Unexpected error in dpsCreateOrGetStore(Zip_Code_Lookup): rc = " +
      (rstring)dpsGetLastStoreErrorCode() + ", msg = " + dpsGetLastStoreErrorString());
} else {
   ...
}
```

Java

```
StoreFactory sf = DistributedStores.getStoreFactory();
Store testStore1 = null;

try {
   testStore1 = sf.createOrGetStore("A_Quick_Store", "ustring", "ustring");
} catch (StoreFactoryException sfe) {
   System.out.println("Unable to create a new store named 'A_Quick_Store': Error code = " +
      sfe.getErrorCode() + ", Error msg = " + sfe.getErrorMessage());
   throw sfe;
}
```

dps store serialization

- Serialize the entire store into a blob and give it to some other entity.

- That some other entity can create a new store and populate it by deserializing the given blob.

Java

```
StoreFactory sf = DistributedStores.getStoreFactory();
Store topBrandsStore = sf.createOrGetStore("2013_Best_Global_Brands_ABC", "int32", "ustring");
// Add few data items.
topBrandsStore.put(1, "Apple");
topBrandsStore.put(2, "Google");
topBrandsStore.put(3, "Coca Cola");
topBrandsStore.put(4, "IBM");
topBrandsStore.put(5, "Microsoft");
topBrandsStore.put(6, "GE");
topBrandsStore.put(7, "McDonald's");
topBrandsStore.put(8, "Samsung");
topBrandsStore.put(9, "Intel");
topBrandsStore.put(10, "Toyota");
// Store serialization.
ByteBuffer serializedStore = topBrandsStore.serialize();

// Store deserialization
sf.removeStore(topBrandsStore);  // Our original store is completely gone.
topBrandsStore = sf.createOrGetStore("2013_Best_Global_Brands_XYZ", "int32", "ustring");
topBrandsStore.deserialize(serializedStore);
System.out.println("This is a list of top 10 rankings for the best global brands in 2013:");
for (KeyValuePair kv: topBrandsStore) {
   System.out.println("'" + kv.getKey() + "' => '" + kv.getValue() + "'");
}

sf.removeStore(topBrandsStore);
```

dps access from non-Streams apps

- Provides a way to share state between Streams and non-Streams apps.
- Full support is already available for external Java applications.
- Work is in progress to provide dps APIs for Python applications.

Python

```
from DpsHelper import *

# Create a store
rc, id = createOrGetStore("Python Test Store1", "boolean", "rstring")
value = "IBM pioneered many of the fundamental hardware and software technologies."

# Put data item into a store
result, rc = put(id, True, value)
if result == True:
        print "1a) Successfully put a data item <boolean, rstring> into a store with an id " + str(id) + "."
else:
        msg = "1a) Unable to put a data item <boolean, rstring> into a store with an id " + str(id) +
            ", Error code = " + str(getLastStoreErrorCode()) + ", Error message = " + str(getLastStoreErrorString())
        print msg

# Get data item from a store
key = True
dummyValue = "Dummy String"
result, rc, value = get(id, key, dummyValue)
print "1b) Data item <boolean, rstring> read from a store with an id " + str(id) + ". key=" + str(key) + ", value='" +
str(value) + "'"

# Remove a store
removeStore(id)
```

Adding new dps backends

- To add a new back-end, one needs to implement a few C++ interfaces
  - Store factory related APIs
  - Store related APIs
  - Iteration related APIs
- These interfaces are completely decoupled from the SPL and Java level details
- They require dealing with keys and values that are byte arrays
- It is a must for a chosen back-end data store to have its own TTL based APIs. Without them, it can't be plugged into the dps DB layer.

dps application patterns

- A few ideas as food for thought:
  - Read application-specific configuration values from a store
  - Common data templates or models used by different application components can be served from a store
  - Hundreds of islands of state information built during a continuous period of analysis can be maintained and shared via the dps (customer behavior pattern detection, Stock portfolio calculation etc.)
  - Analytic engines can keep their intermediate results in a store without worrying about data loss during recovery after a PE crash
  - Massive reference data look-up needed in a multitude of operators can be kept and read from a store (e-g: Geohash, customer profile in a Telco CDR application etc.)
  - Memory intensive bloom filter map can be kept in a store. (e-g: Telco CDR applications).
  - Wherever there is a need for coordinated transactional activities to be performed by many distributed components, distributed locking (dl) can be of use.

dps technology assets thus far

- DPS toolkit is available in the IBMStreams github as well as in the Streams product.
  - https://github.com/IBMStreams/streamsx.dps

- IBM developerWorks technical article about the dps:
  - http://tinyurl.com/nxrf3gg

Possible future extensions (dps)

- Currently supported back-end data stores (memcached, Redis etc.) are a good start for some applications.
- IBM Research HydraDB project promises an unmatchable high performance advantage. That will be added as another supported back-end data store.
- TOAD (Trillion Operations A Day) is an ambitious Flash SSD based persistent data store research work in Almaden. This is another candidate for a specialized dps back-end data store.

Acknowledgements

- Martin Hirzel (His ideas helped in refining our store layout design)
- Kun-Lung Wu (His interim reviews were tremendously helpful)
- HydraDB research work in the Watson lab
  - Rashed Bhatti
  - Xavier Guerin
  - Yuqing Gao
- Early interest from customers (as of Nov/2013)
  - Financial risk and news analytics firm in NYC
  - A majestic Government in an European country
  - A Telco giant deeply rooted in the north eastern United States
  - A large wireless carrier in India
  - Friends in the IBM Unica product suite
  - A few freelance Streams developers

Demo

If time permits, show running code now.

Thing1_Store

Ticker	Company
IBM	IBM Corp.
F	Ford Motor Co.
BA	The Boeing Co.
T	AT&T Inc.
CSCO	Cisco Systems
GOOG	Google Inc.
INTC	Intel Corp.

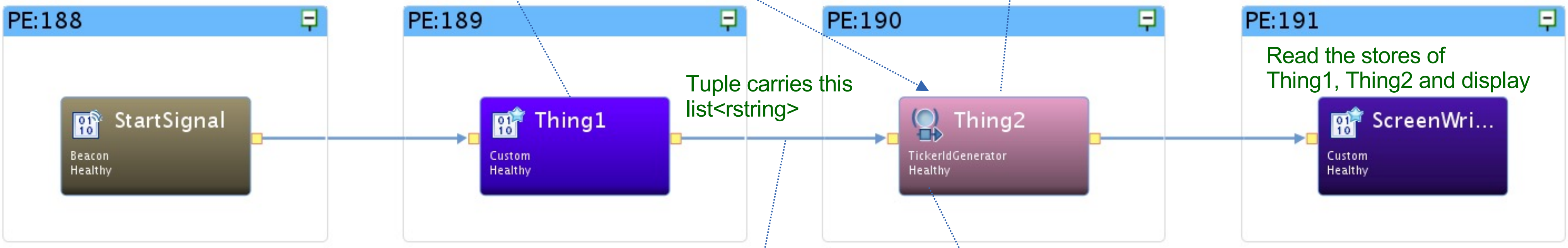
Thing2_Store

	Unique ID
IBM	???
T	???
GOOG	???
BA	???

Thing2 writes here

Thing2 reads from here and computes unique ticker ids

Thing1 writes here



Tuple carries this list<rstring>

Read the stores of Thing1, Thing2 and display

["IBM", "T", "GOOG", "BA"]

C++ or Java operator