# CAPSULE: Language and System Support for Efficient State Sharing in Distributed Stream Processing Systems

Giuliano Losa♦1 Vibhore Kumar♠2 Henrique Andrade♠3* Buğra Gedik♣4
Martin Hirzel♠5 Robert Soulé♠6† Kun-Lung Wu♠7
♦Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
♠Thomas J. Watson Research Center, IBM Research, 19 Skyline Drive, Hawthorne, NY 10532, USA[2]
♣Department of Computer Engineering, Bilkent University, Bilkent, Ankara, 06800 Turkey
[1]giuliano.losa@epfl.ch,{[2]vibhorek,[5]hirzel,[7]klwu}@us.ibm.com,{[3]henrique.c.m.andrade,[4]bgedik}@gmail.com,[6]soule@cs.nyu.edu

## ABSTRACT

Data stream processing applications are often expressed as data flow graphs, composed of operators connected via streams. This structured representation provides a simple yet powerful paradigm for building large-scale, distributed, high-performance applications. However, there are many tasks that require sharing data across operators, and across operators and the runtime using a less structured mechanism than point-to-point data flows. Examples include updating control variables, sending notifications, collecting metrics, building collective models, etc. In this paper we describe CAPSULE, which fills this gap. CAPSULE is a code generation and runtime framework that offers an easy to use and highly flexible framework for developers to realize shared variables (CAPSULE term for shared state) by specifying a data structure (at the programming-language level), and a few associated configuration parameters that qualify the expected usage scenario. Besides the easy of use and flexibility, CAPSULE offers the following important benefits: (1) Custom Code Generation - CAPSULE makes use of user-specified configuration parameters and information from the runtime to generate shared variable servers that are tailored for the specific usage scenario, (2) Composability - CAPSULE supports deployment time composition of the shared variable servers to achieve desired levels of scalability, performance and fault-tolerance, and (3) Extensibility - CAPSULE provides simple interfaces for extending the CAPSULE framework with more protocols, transports, caching mechanisms, etc. We describe the motivation for CAPSULE and its design, report on its implementation status, and then present experimental results.

## Categories and Subject Descriptors

B.3.2 [**Design Styles**]: Shared Memory; D.2.11 [**Software Architectures**]: Data Abstraction

*Currently employed by Goldman Sachs.
†Currently student at New York University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*DEBS '12,* July 16–20, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-1315-5 ...$10.00.

## Keywords

Distributed Shared State, Stream Processing, Consistency Models

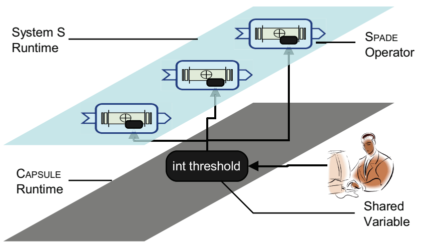
## 1. INTRODUCTION

Distributed data stream processing systems [3, 1] often provide an abstraction of a data-flow graph, composed of operators and connected via streams, to express stream processing applications [19]. The data-flow graph based representation provides a simple but powerful paradigm for building large-scale, distributed and high-performance applications. While traditional wisdom warns against the use of shared state in distributed stream processing systems that strive for high-performance and scalability, there are data stream processing applications that can benefit from the existence of a state sharing mechanism. Example usage scenarios, shown in Figure 1, include externally accessible 'control' variables that affect the behavior of operators contained in a streaming application, operators that need to efficiently persist their state to facilitate restart in event of a failure, or shared data structures with a usage specific consistency model that are accessed by multiple operators.

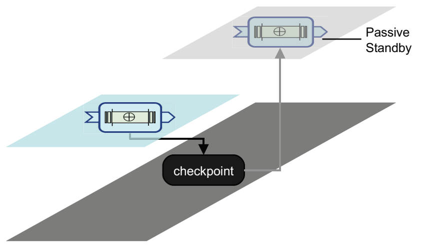
These usage scenarios are derived from our experiences with System S [3] − a large-scale, distributed data stream processing middleware, which has been under development at the IBM T. J. Watson Research Center. Due to the lack of system-supported abstractions for state sharing, such functionality has been painstakingly implemented using custom written code by many applications. It is evident that *i*) there are usage scenarios that can benefit from the existence of a state sharing mechanism, and *ii*) the users, in absence of such a mechanism, will implement custom workarounds that might not only be cumbersome and complex but might be inefficient too.

Implementing a state sharing mechanism for a high-performance data stream processing system like System S poses a very unique set of challenges and opportunities. System S, with its declarative programming language SPADE [13], offers an easy to use interface for expressing a wide-range of streaming applications. As a result, the design of the state sharing mechanism has to maintain the same level of *ease of use* and provide *flexibility* for handling a wide range of usage scenarios. Another important requirement stems from the needs of the stream processing applications, and these include *scalability*, *high-performance* and *fault-tolerance*. To deal with these challenges, besides a careful design, one can exploit the ability of stream processing applications to tolerate a range of *relaxed consistency guarantees*. We now describe these requirements and opportunities in some detail:

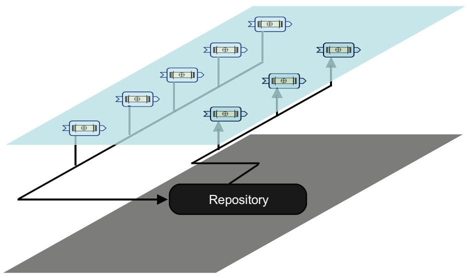
- *Ease of Use & Flexibility* - Many of the users of System S creating stream processing applications are domain experts and an-



(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.

**Figure 1: Example usage scenarios**

alysts, with sufficient but not a deep understanding of issues related to management of distributed shared state. To not adversely impact the usability, and to accommodate the needs of the wide cross-section of stream processing applications, the shared state implementation should not only be easy to use but should also be flexible.

- *Scalability, High-Performance & Fault-Tolerance* - Stream processing applications operate on large volumes of rapidly arriving data. In some cases the shared state may have to be shared with a number of other stream processing operators, or it may have to handle a large number reads or writes per second, or even provision for replication of shared state. These usage scenarios are representative of the applications that will make use of shared state and require the state sharing mechanism to scale, offer high-performance and be able to tolerate faults in some scenarios.
- *Relaxed Consistency Guarantees* - Given the challenges involved in implementing state sharing for stream processing applications, in some usage scenarios, it would be impossible to offer the desired levels of scaling and performance without relaxing the consistency guarantees. It is not surprising that many stream processing applications can tolerate relaxed consistency guarantees; and our state sharing mechanism should be able to exploit such relaxations.

## 1.1 Existing Solutions

Traditionally, developers have relied on message passing, distributed shared memory or even database systems for enabling state sharing in distributed systems. In general, distributed systems that are based on message passing tend to offer better performance due to the application specific nature of the implementation. However, the application specific nature of the system also implies more lines of custom code, which in turn makes this paradigm more cumbersome and error-prone. Message passing systems generally fail to offer the ease of use that users have come to expect from System S and SPADE. Distributed shared memory, on the other hand, offers an easy to use abstraction of the underlying distributed memory and hides several protocol and transport level details from the developer. However, as would be expected from a generic implementation, it suffers from lack of scalability and inadequate performance due to its inability to exploit application level knowledge. This lack of scalability, inadequate performance and inflexibility rules out off the shelf distributed shared memory systems. Finally, Database systems lack the performance that is needed for many distributed applications. This is because database systems provide more functionality than needed, resulting in performance overheads. It is amply clear that the existing solutions for state sharing are not the right fit for use in distributed data stream processing systems. The following examples illustrate some of the usage scenarios for a state sharing mechanism in distributed stream processing systems.

EXAMPLE 1. *Many applications need their stream operators to be dynamic.* They may need to control the way these operators process and/or route the incoming data stream. Consider an application like the one shown in Figure 1, which might be filtering some streaming data to extract a subset that is of interest to the user. For efficiency and scalability reasons, a large number of filter operators might be deployed on many machines. In long running stream processing applications a user may occasionally want a means to modify the filtering criteria associated with the operators. This scenario requires externally accessible shared state that is tuned for frequent reads and infrequent writes.

EXAMPLE 2. *In a stream processing application, with potentially thousands of operators, it is not uncommon to have operator failures.* In many cases the operators contained in a streaming application cannot tolerate to lose the accumulated data because of a fault in the system. A way for providing operator fault tolerance is to regularly checkpoint their state to be able to restore it elsewhere if the processing node supporting the operator fails. By using a replicated implementation of shared state, every operator could periodically write snapshots of its state to a shared object. The System S runtime would access this replicated state to restore an operator and restart it. Providing this very general primitive would ease operator fault tolerance implementation in all applications.

EXAMPLE 3. *Some stream processing applications process data by making use of a complex analytic model.* The model itself is highly dynamic and should be continuously adjusted as more data is observed. A concrete example is predicting failures in a semiconductor fab. The fab has numerous sensors measuring physical parameters associated with its different units. By processing the flux of information from those sensors, one can detect patterns that lead to failure of some components with a high probability, but those patterns are highly sensitive to changes in the environment that cannot be controlled. The data from the sensors has thus to be continuously processed and correlated to failures in order to adapt the model to the changing conditions. For performance and scalability reasons the model could be built cooperatively by multiple operators and used by numerous others. Devising a wiring scheme with streams would be very complex in this case, whereas using the SPADE type system to represent the model as one shared object would render the task simple, allowing the programmer to focus on application level algorithms instead of data dissemination issues.

### 1.2 Contributions

In this paper we present CAPSULE, a code generation and runtime framework that provides IBM's System S stream processing middleware the capability to efficiently share state across multiple runtime entities including data-flow operators, daemons and runtime console under a range of usage scenarios. In the remainder of this section we outline the main contributions of this work:

- *Language-Level Constructs & Seamless Access* - CAPSULE offers constructs at SPADE language-level to declare shared variables, specify their visibility characteristics and specify configuration parameters like fault-tolerance requirements, expected read/write ratio, expected variable size (in case of dynamic sized shared variables), etc. to customize the variable for the particular usage scenario. Once a shared variable has been declared it can be used to seamlessly access the shared state, treating the variable as a regular SPADE data-type.
- *Optimized Custom Code Generation* - CAPSULE makes use of code generation to embed suitable protocol, transport, caching mechanism, etc. into the implementation of each shared variable. These customizations are based on the user-specified configuration parameters and the information gathered from the runtime system.
- *Capability to Compose Variables* - Internally, to achieve desired levels of scalability, performance and fault-tolerance, CAPSULE makes use of its capability to hierarchically compose variables into groups of variables that are related using a specific protocol. For instance, a single shared variable visible to the user might in fact be composed of three causally related variables, and each such causally related variable might in turn be implemented as two atomically related variables.
- *Extensible Architecture* - CAPSULE architecture contains well-defined interfaces for implementing more consistency protocols, transport mechanisms and caching mechanisms. This is an important attribute of our architecture because it is not possible to envision all the possible usage scenarios and we cannot select and restrict the system to particular protocol, transport or caching mechanisms.

## 2. BACKGROUND: SYSTEM S & SPADE

System S is a large-scale, distributed data stream processing middleware under development at the IBM T. J. Watson Research Center. It supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes. The System S runtime can execute a large number of long-running applications that take the form of data-flow graphs. A data-flow graph consists of a set of Processing Elements (PEs) connected by streams, where each stream carries a series of Stream Data Objects (SDOs). The PEs are basic execution containers that are distributed over compute nodes and a node is generally host to multiple PEs. The compute nodes are organized as a shared-nothing cluster of workstations (COW) or as a large supercomputer (e.g., Blue Gene). The PEs communicate with each other via input and output ports, which are connected by streams.

SPADE (Stream Processing Application Declarative Engine) is the declarative stream processing engine of System S. It is also the name of the declarative language used to program applications. The language is used to express parallel and distributed data-flow graphs containing the operators and resulting streams required to carry out the actual processing for an application. SPADE currently offers toolkits of type-generic built-in stream processing operators. It is also possible to augment a toolkit with additional user-defined built-in operators (UBOPs) or create new toolkits altogether. The SPADE language also features a broad range of stream adapters that can be used to ingest data from outside sources and publish data to outside destinations, such as network sockets, relational and XML databases, file systems, as well as several proprietary platforms. SPADE makes use of code generation to fuse operators into processing elements so as to match application characteristics such as communication patterns among the operators as well as processing capabilities of a particular compute node. This code generation approach is extremely powerful because through simple recompilation one can go from a fully fused application to a fully distributed one, adapting to different ratios of processing to I/O provided by different computational architectures (e.g., blade centers versus Blue Gene).

## 3. CAPSULE OVERVIEW

Architecturally, CAPSULE contains two, more or less independent sub-systems - one interacts with the SPADE compiler to generate custom and optimized code for realization of shared variables and the other interfaces with the System S runtime infrastructure to assist in deployment and management of the shared variables at runtime. Figure 2 shows the interaction of the two CAPSULE components, labeled *CAPSULE Code Generator* and *CAPSULE Daemon* with the remainder of System S.

A typical System S application developer will make use of the SPADE language to specify the data-flow, and the associated shared variables with their corresponding configuration parameters. The SPADE program is then submitted to the compiler, which along with the generation of System S artifacts (like the processing elements) also generates a model that describes the shared variable structure and configuration parameters. The CAPSULE code generator makes use of the model supplied by the SPADE compiler and the runtime information determined by CAPSULE daemons to generate the *shared variable servers*, the corresponding *shared variable description language* or SVDL file and the *shared variable stubs* to enable access to the shared variable.

The shared variable servers are compiled into DLLs (dynamic link libraries), and encapsulate a server-side transport and protocol object, and a data object that is either compiled into the shared vari-

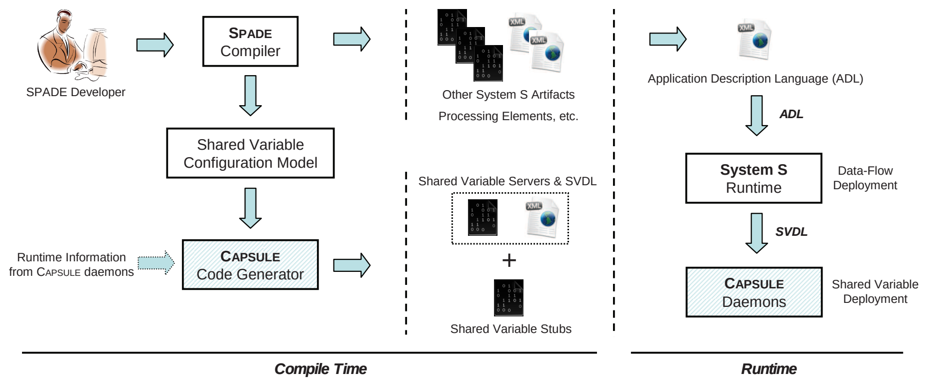

SPADE Developer
SPADE Compiler
Shared Variable Configuration Model
Runtime Information from CAPSULE daemons
CAPSULE Code Generator
Other System S Artifacts
Processing Elements, etc.
Shared Variable Servers & SVDL
Shared Variable Stubs
Application Description Language (ADL)
ADL
System S Runtime
Data-Flow Deployment
SVDL
CAPSULE Daemons
Shared Variable Deployment
Compile Time
Runtime

**Figure 2: Interaction of CAPSULE code generator and daemons with System S components**

able server or is a reference to a data object that is remote. To enable remote access and composability, the shared variable exports a remotely accessible data object interface. The SVDL file contains references to the shared variable server DLLs, contains the variable name, expresses their composition and also contains the location where the shared variable server DLLs will be loaded at runtime. Finally, the shared variable stubs expose an access interface that is exactly the same as the regular SPADE data types. The stubs besides the interface code also contain a client-side transport and protocol object, and may also contain a cache object. More details about these artifacts will be provided in Section 4.

At deployment time the SVDL file is submitted to any of the several CAPSULE daemons that may be running on the cluster nodes. A single CAPSULE daemon is designated as the owner of the shared variable described in the SVDL file, which then contacts other CAPSULE daemons to deploy the shared variable. Any System S entity which wants to access the shared variable makes use of the stub to do so.

## 4. DETAILED DESIGN

In this section we describe the detailed design of the CAPSULE framework, and provide details about the interesting design choices that have been made for the framework. We start by describing language level features that enable the shared variables, describe the CAPSULE code generation framework, shared variable servers, shared variable description language, and describe the CAPSULE daemon.

### 4.1 Shared Variable Declaration and Usage

An important feature of the shared variables provided by the CAPSULE framework is that the shared state can be declared and accessed as a regular SPADE data type. Furthermore, a user can provide hints to the CAPSULE framework using the configuration parameters to assist in generation of code that is customized for a particular usage scenario. These two features of the CAPSULE framework are exposed to the end user as language features in the coming version of the SPADE language.

The syntax for shared variable declaration in SPADE and a corresponding example is shown in Figure 3. A shared variable, besides the type declaration, is qualified by modifiers and configs. Modifiers specify a particular behavior that is enforced in the language, whereas configs are passed to the underlying implementation, which has the freedom to decide what to do with them. There are three possible modifiers. They specify how to instantiate a shared variable (*static* keyword), its visibility to other operators and applications (*public* keyword), and if it is mutable or immutable (*mutable* keyword). The static modifier specifies that all instances of the operator defining a variable will share the same copy. Without this modifier, every operator instance will have its own copy of the variable. The public modifier affects the visibility of the shared variable. By default (without the public modifier), a shared variable is visible only in the operator in which it is declared. If the public modifier is used, the shared variable will be visible in the whole system. This means that it can be referred to anywhere in the same application or in another application running in the System S instance. Only static variables can be declared public, and they can be accessed by providing their name, the name of the operator defining them and the SPADE namespace of the application. The mutable modifier allows a shared variable to be modified, by default it is read only. Shared variable optionally have an initializer. If they are read only, the initializer is mandatory. The SPADE compiler statically enforces the restrictions specified by the modifiers. Once declared, a shared variable can be used as any other SPADE variable, except that the restriction specified by the modifier have to be followed (if not, the program will not compile).

Single operations on shared data types are ordered by the underlying implementation. They can be totally ordered in case the user chooses an atomic consistency model. In near future, we plan to provide language level constructs for supporting *locks*, which can then be used for coordinating updates across multiple shared variables.

|  |  |  |
|---|---|---|
| *varDef* | ::= | *varModifier*$^*$ *type ID* ( ‘=’ *expr* )$^?$ *varConfigs* |
| *varModifier* | ::= | ‘`public`’ \| ‘`static`’ \| ‘`mutable`’ |
| *varConfigs* | ::= | ‘`;`’ \| ‘`{`’ ‘`config`’ *config*$^+$ ‘`}`’ |
| *config* | ::= | *ID* ‘`:`’ *expr* ‘`;`’ |

```
public static mutable list<int32> s_l {
    config
        consistency : causal;
        sizeHint : 1GB;
        writesPerSecond : 5;
        readsPerSecond : 500;
}
```

**Figure 3: Syntax and example of a shared variable declaration**

### 4.2 CAPSULE Objects

The CAPSULE code generator relies on a set of objects, that it composes together, to generate the shared variable servers and the shared variable stubs. Our design for these generated artifacts, shown in Figure 4, segregates the functionality into four main objects - the data object, the protocol object, the transport object and the cache object, and a carefully defined interaction interface - the invoke interface that enables the composition of these objects. For instance, the code generator, using our design, can generate a shared variable server for an int32 data type that is encapsulated in a data object, with a protocol object that enforces atomic consistency between the replicas of this data object and using a CORBA [10] based transport object. The design provides us the capability to choose an implementation of the object that is suitable for a specified usage scenario. Another interesting aspect of our design is the fact that a shared variable server can also act as a server-side data object. This allows us to compose shared variables servers that may use a set of other shared variables servers as data objects. This composition capability exploited using the shared variable description language (described in Section 4.3.3) is critical to the scalability, performance and fault-tolerance provided by the CAPSULE framework. We start by describing a simple invoke interface that is implemented as a composition mechanism by multiple CAPSULE objects and then provide details about the four CAPSULE objects.

#### 4.2.1 invoke *Interface*

The invoke interface consists of a single method that can be used for invoking any operation on the shared data structure. This interface is internal to CAPSULE but is central to the composability of the CAPSULE objects, and allows us to efficiently invoke operations on user-defined shared nested data structures. The interface method takes the following form - `void invoke(int methodIndex, Buffer inParams, Buffer outParams);`

#### 4.2.2 *Data Object*

The CAPSULE data object comes in two flavors - one for the server side and the other for the stub side. The server-side data object encapsulates the data structure that the user wants to share and implements the invoke interface for allowing the invocation of operations on the shared data-structure. We also define a server-side data object reference stub, which is a remote interface to the server-side data object. The server-side data object translates and invoke call to operations on the shared data-structure, while the reference stub forwards the invoke call to a remote server-side data object. Both, the server-side data object and the data object reference stub, also expose a set of three helper methods for the protocol object, these include - (1) getInvocationScope method, which is used by the protocol object to determine whether an invoke call identified by methodIndex should be invoked on all data objects participating in the protocol, or should be invoked on only one such data object, (2) splitParams method, which is applied to inParams when an invoke call is determined to be an invoke-on-all call; it returns the inParams to be forwarded with the resulting invoke calls, and (3) mergeParams method, which is applied to outParams returned by each individual invoke call, after an incoming invoke call is determined to be an invoke-on-all call. Since the shared variable server also implements the invoke interface, therefore when used with the server-side data object reference, it can be treated like a server-side data object.

The stub-side data object exposes the interface that is visible to the end user and it is the same interface that a user would use when accessing a regular SPADE data-type. The stub-side data object translates the shared variable data access to an *invoke* call on the stub-side protocol object.

#### 4.2.3 *Protocol Object*

The protocol objects at the server-side and the stub-side together implement a specific mechanism for calling the invoke method on the participant, which in this case are the server-side data objects or the server-side data object reference stubs.

The server-side protocol object exposes the invoke interface to the server-side transport object. The protocol object essentially accepts and invoke call and determines how and to which of the participating server-side data objects or data object reference stub the call should be forwarded to. On receiving an invoke call, the protocol object, uses the methodIndex and the server-side data object's or the data object reference stub's getInvocationScope method to determine if the invoke method should be invoked on all the participating data objects, or should be invoked on only one of the participating data objects. For instance, consider a case where an Integer shared variable that is replicated three times for fault-tolerance, and the replicas have to be atomically updated. In this case an invoke method that corresponds to getting the value needs to be invoked on just one of the replicas, while the set call will have to be invoked on all the replicas. Now, if the invoke method needs to be invoked on all the replicas then the protocol object uses the splitParams and mergeParams methods provided by the data object to split the input parameters for invoke calls, and merge the return values from all the invoke calls, respectively. The splitParams and mergeParams are typically useful when using a partitioned shared list, for instance.

The stub-side protocol object is a component that doesn't occur in many realizations of the shared variable stub. In certain cases, having a stub-side protocol object can optimize the performance of the shared variable. For example, consider the partitioned version of the shared list that was described earlier in this section. A user invoking an operation to retrieve a particular element from the list might contact a partition that may not have the element available. The contacted partition then forwards the invocation to the appropriate partition and the element is delivered to the user. When using a stub-side protocol object, the task of resolving the partition can be done by the stub-side protocol object and can result in avoiding an additional network round-trip. The stub-side protocol exposes the same invoke interface to the stub-side data object, maintains information about the server-side data object and uses the invoke method exposed by the stub-side transport object.

The protocol objects contain certain members and/or properties that have to be initialized at deployment time. These members include the references for any participating remote objects, and protocol specific initialization like designating a master for a master slave protocol, etc. This initialization is carried our by the CAPSULE daemon by remotely invoking (using CORBA) methods on the newly created protocol object.

#### 4.2.4 *Transport Object*

The transport object serves the purpose of hiding and encapsulating the details of transport mechanism used for propagating the invoke call from the stub to the server. The transport objects provide the CAPSULE architecture the capability to adopt and use possibly any mechanism that can invoke operations on a remote object, as long as the implementation conforms to the invoke interface specification. A developer can easily implement this interface for the server and the stub sides and start using a new transport for the CAPSULE shared variables.

The server-side transport object implements an additional interface method that is used by the CAPSULE daemon to retrieve a stringified reference to the transport object. This reference is used by the stub-side transport object to establish connection with the server-side transport object.

The stub-side transport object exposes a connect method which accepts a stringified reference and establishes connection with the

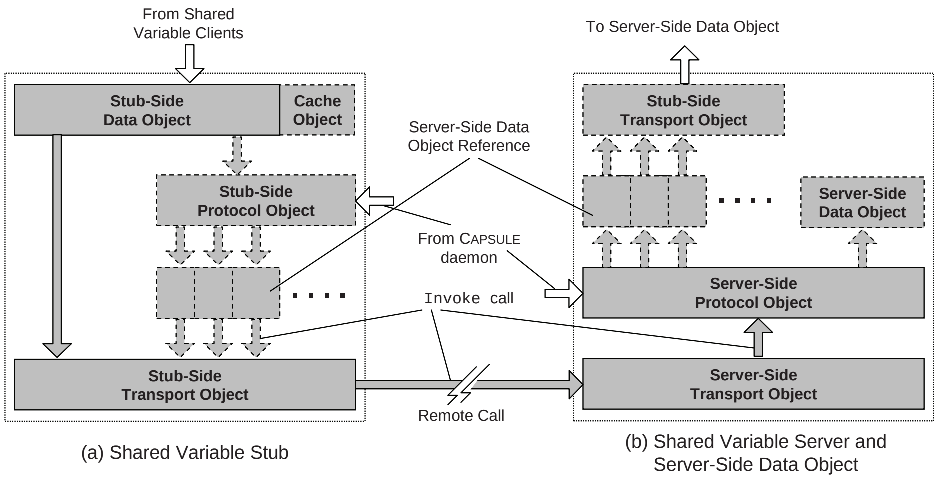

From Shared Variable Clients
Stub-Side Data Object
Cache Object
Stub-Side Protocol Object
Stub-Side Transport Object
(a) Shared Variable Stub
Server-Side Data Object Reference
From CAPSULE daemon
Invoke call
Remote Call
To Server-Side Data Object
Stub-Side Transport Object
Server-Side Data Object
Server-Side Protocol Object
Server-Side Transport Object
(b) Shared Variable Server and Server-Side Data Object

**Figure 4: Architecture of the Shared Variable Stub and the Shared Variable Server**

server-side. This method is used by the stub-side data object and by server-side data object reference stubs.

### 4.2.5 *Cache Object*

The cache object is an optional component that can exist alongside the stub-side data object. If present, it intercepts the `invoke` call and returns the value contained in the cache. The cache contents can be updated using a pull or a push based method. For the pull based methods, the implemented CAPSULE cache object supports fetching of data from a shared variable server using a time or count based policy. When using the time based policy the cache is updated every `t` time units. For a count based policy the cache is updated after every `n` number of invocations of the `invoke` method. For the push based method the cache object exposes a remotely accessible `update` interface, which is used by the shared variable server to push the updates. The decision to have or not have a cache object is not directly exposed to the user but is determined by the CAPSULE code generator. For instance, a scenario with a high read to write ratio might be a good candidate for using a cache object.

## 4.3 CAPSULE Code Generator

The SPADE compiler generates a model called the Shared Variable Configuration Model (SVCM) corresponding to each shared variable declared in the SPADE program. This model and the information about the runtime system as determined by the CAPSULE daemons is then used by the code generator to generate the Shared Variable Servers, the Shared Variable Stubs, and the Shared Variable Description Language (SVDL) file. The CAPSULE code generator makes extensive use of the template meta-programming capabilities provided by the C++ language. We now provide a detailed description of the generated artifacts and the configuration driven customized code generation framework.

### 4.3.1 *Shared Variable Server*

The shared variable server, as shown in Figure 4, contains a server-side transport object, a server-side protocol object and may contain a server-side data object or a set of server-side data object reference stubs or both. An interesting architectural feature of the shared variable server is the fact that it can also be a server-side data object. This feature allows for the composition capabilities, where a shared variable server can refer to multiple underlying shared variable servers (possibly at remote locations). The composability provided by the shared variable server plays an important role in achieving scalability and fault-tolerance for the shared variables.

The code generator is capable of generating two distinct versions of the shared variable server. In one case the server-side data object is embedded inside the code, while in the other case it is not. The latter is used in cases when the shared variable server has only references to the server-side data objects, and is called a *blank* shared variable server. These references are populated by the CAPSULE daemon at deployment time. To create a shared variable server or a blank shared variable server the code generator uses the simple templatized objects provided by the CAPSULE framework:

```
CapsuleServer<Transport,Protocol,DataObject>
CapsuleBlankServer<Transport,Protocol,DataObject>
```

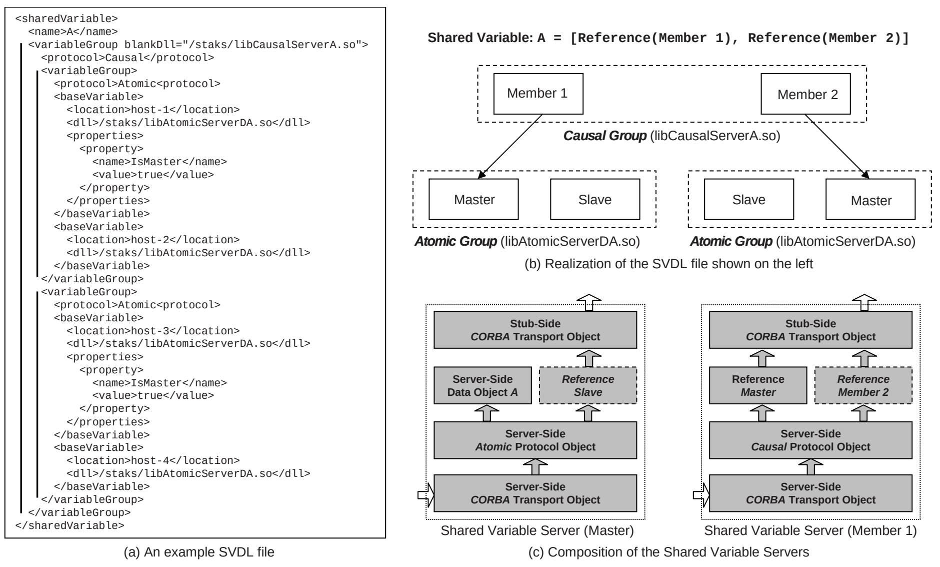
### 4.3.2 *Shared Variable Stub*

The shared variable stub is the component that is used by the System S entities like operators, daemons, etc. to gain access to the shared variable. Shown in Figure 4, the shared variable stub consists primarily of a stub-side data object and a stub-side transport object. In certain cases, which may benefit from the knowledge of the underlying shared variable servers (e.g. in case of partitioning protocol), the generated shared variable stub may also contain the stub-side protocol object and a set of server-side object reference stubs to resolve the appropriate shared variable server. Occasionally the shared variable stub may also contain a cache object to exploit the scenarios which exhibit a high read-to-write ratio, or scenarios which can tolerate stale data. The following templatized objects are used as a way to generate custom shared variable stub:

```
CapsuleStub<Transport,DataObject,Cache>
CapsuleProtocolStub<Transport,Protocol,DataObject,Cache>
```

### 4.3.3 *Shared Variable Description Language*

The shared variable description language or the SVDL is an XML-based representation that is used for specifying the composition of a single shared variable in terms of shared variable servers. This capability to compose shared variable servers can be used to fine-tune the performance, scalability and fault-tolerance attributes associated with a shared variable. For instance, using SVDL one can fine-tune the number of replicas of a shared variable server that will be deployed to achieve desired level of fault-tolerance. Consider another scenario, where a user would like to have performance and



```
<sharedVariable>
  <name>A</name>
  <variableGroup blankDll="/staks/libCausalServerA.so">
    <protocol>Causal</protocol>
    <variableGroup>
      <protocol>Atomic<protocol>
      <baseVariable>
        <location>host-1</location>
        <dll>/staks/libAtomicServerDA.so</dll>
        <properties>
          <property>
            <name>IsMaster</name>
            <value>true</value>
          </property>
        </properties>
      </baseVariable>
      <baseVariable>
        <location>host-2</location>
        <dll>/staks/libAtomicServerDA.so</dll>
      </baseVariable>
    </variableGroup>
    <variableGroup>
      <protocol>Atomic<protocol>
      <baseVariable>
        <location>host-3</location>
        <dll>/staks/libAtomicServerDA.so</dll>
        <properties>
          <property>
            <name>IsMaster</name>
            <value>true</value>
          </property>
        </properties>
      </baseVariable>
      <baseVariable>
        <location>host-4</location>
        <dll>/staks/libAtomicServerDA.so</dll>
      </baseVariable>
    </variableGroup>
  </variableGroup>
</sharedVariable>
```

Shared Variable: A = [Reference(Member 1), Reference(Member 2)]
Member 1
Member 2
Causal Group (libCausalServerA.so)
Master
Slave
Slave
Master
Atomic Group (libAtomicServerDA.so)
Atomic Group (libAtomicServerDA.so)
Stub-Side CORBA Transport Object
Server-Side Data Object A
Reference Slave
Server-Side Atomic Protocol Object
Server-Side CORBA Transport Object
Shared Variable Server (Master)
Stub-Side CORBA Transport Object
Reference Master
Reference Member 2
Server-Side Causal Protocol Object
Server-Side CORBA Transport Object
Shared Variable Server (Member 1)

(a) An example SVDL file

(b) Realization of the SVDL file shown on the left

(c) Composition of the Shared Variable Servers

**Figure 5: Shared Variable Description Language**

fault-tolerance, but is ready to sacrifice atomic consistency; here the SVDL can be used to specify the deployment configuration shown in Figure 5, which specifies a shared variable with two causally related shared variable servers (for performance), and each causally related shared variable server in turn contains two atomically related shared variable servers (for fault-tolerance).

The SVDL defines three important entities: (1) a base variable, which is represented as `baseVariable` in the SVDL identifies a shared variable server that has a server-side data object embedded into it. The deployment of a base variable results in a single remote reference being returned, (2) a variable group represented as `variableGroup` corresponds to a protocol, which is enforced on the members of the variable group. A variable group can consist of any number of base variables, variable groups or shared variables and its deployment results in a set of references being returned, and finally (3) a shared variable represented using `sharedVariable` in the SVDL can consist of a single variable group or a single base variable and has a name associated with it. The user can retrieve a reference to this object by querying the CAPSULE daemon for the specified name. Each group can have properties associated with them, which are used by the CAPSULE daemon to initialize the shared variable server. An example of this property is the `IsMaster` property associated with a variable group that uses an atomic master-slave protocol.

### *4.3.4 Configuration-Driven Code Generation*

There are two levels of customization that are possible with the CAPSULE framework. The first one relates to the generation of code for the shared variable servers and the shared variable stubs, while the second one focuses on the customizations that could be done via the SVDL associated with a shared variable.

For specification of user preferences, the currently supported configuration parameters at the language level include `sizeHint`, `consistency`, `faultTolerance`, `readsPerSecond` and `writesPerSecond`. The values for these parameters, if specified, are available to the code generator via the SVCM. For runtime information, we utilize the CAPSULE daemons to gather data about the performance of each protocol on the target deployment system. This performance data for each protocol is a table containing information about maximum updates per second achieved for different discrete values of the following parameters: number of replicas, number of clients and read-write ratio.

To determine the protocol, transport and inclusion or exclusion of cache, and to generate the SVDL, the current implementation of CAPSULE makes use of a rule-based approach for the purpose of generating code, and is still under development.

## 4.4 CAPSULE Daemon

The CAPSULE daemon is the runtime component that assists in deployment of the shared variables. The daemon is assumed to be running on the nodes which host a System S instance. Any CAPSULE daemon is capable of accepting a SVDL file for deployment of a shared variable. Once a shared variable has been deployed, the responsibility of maintaining a reference to the deployed shared variable is assigned to a CAPSULE daemon using a CHORD [20] like lookup mechanism, where the name of the variable is treated as the key. The shared variable servers run in the context of these daemons. System S provides the capability to monitor and restart the CAPSULE daemons. However, we have not yet implemented any functionality that attempts to save the state maintained by the CAPSULE daemons. A proposal that we are working on for stateful recovery includes using shared variables for maintaining the state of the CAPSULE daemons.

## 5. IMPLEMENTATION STATUS

CAPSULE is being implemented in C++, and currently we have a codebase of around 6000 lines of code. The basic flow that leads from SVCM to SVDL is in place and we currently support primitive data types and collections of primitive types (maps, lists and hashes) as shared variables. The support for arbitrary user-defined types as shared variables is still under development, but we do not foresee any significant challenges in achieving the same. The CAPSULE daemon is almost fully functional, except for the task of stateful recovery from a failure. We currently support CORBA as the transport for implementing the transport object. As for the protocol objects, we have three different implementations for protocols that enforce consistency, and a fourth protocol which enables partitioning for large data collections. We provide a brief description of the implemented protocol objects:

- *Atomic Master-Slave Protocol (AMS)* - AMS is a protocol that enforces atomic consistency on the participating server-side data objects. It optimistically returns from the bottleneck `set` calls, and is therefore able to receive another `invoke` call while the master is still updating the slaves. It performs well for small number of clients
- *Atomic Master-Slave Buffer Protocol (AMSB)* - AMSB, again, is a protocol that enforces atomic consistency on the participating server-side data objects. The protocol keeps buffering and blocking any calls that it may receive while the master is updating the slaves in response to a `set` call. On returning from the call, the master returns all `get` calls immediately with the current value of the variable and then applies and propagates only the final `set` call. The protocol is not suited for small number of clients due to overheads caused by buffering, but performs well for large number of clients.
- *Causal Protocol* - The causal protocol is implemented using vector clocks and maintains causal consistency between the participating shared variable servers. Here, each client can interact only with one designated replica and causality is established by exchanging messages between the participants on a best effort basis or at configurable time intervals. The causal protocols net throughput for `invoke` calls increases as the number of replicas are increased, as opposed to the decrease encountered by atomic protocols mentioned above.
- *Partitioning Protocol* - The partitioning protocol doesn't enforce any consistency model on the participating server-side data objects, but rather provides a way to load-balance and provide scalability for large data collections like `list`.

We also have an implementation of the cache object, which in case of push based caching provides us the capability to subscribe to updates from a shared variable server. In case of pull based caching, the cache object can be configured to fetch updates from a shared variable server at regular time intervals, to fetch updates from a shared variable server in response to every `n`th `invoke` call. The cache object doesn't still support the caching of large collections of data.

## 6. EVALUATION

In this section we report on the experimental evaluation of the CAPSULE framework at achieving its goal of providing efficient access to shared state. To establish the baseline, we start by evaluating the performance of a shared variable implemented as a single barebones CORBA server for a varying number of remote clients. Thereafter, we evaluate the performance of shared variables using four protocol objects that have been implemented, so far, for the CAPSULE framework. In particular, we evaluate the performance of primitive shared variables (32 bit integer) with AMS, AMSB and the Causal protocol for varying number of clients and server replicas. We also evaluate the effect of number of partitions and clients on our partitioning protocol when using a `list` data type. The following experiments were conducted on nodes with 2 x dual core Intel 3.0 Ghz CPUs with 8 GB RAM and 2 x 73.4 GB hard disks.

### 6.1 Establishing Baseline

We implemented a CORBA server encapsulating a 32 bit integer, to which several clients connect and send their updates. The server maintains a copy of the shared data guarded by a lock. All updates are processed in a total order by this central server. This evaluation

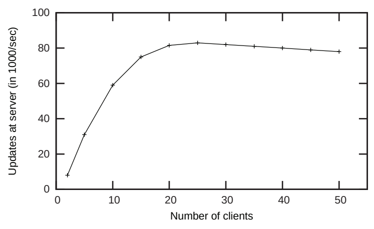


**Figure 6: Variation of update rate with number of clients using a CORBA based single server implementation**

provides us an idea about the performance to be expected from the CAPSULE shared variable implementation, and establishes a baseline for performance comparison. The results shown in Figure 6 show that the net throughput achieved with a single CORBA server steeply increases from 8000 updates/sec for 1 client to ≈80,000 updates/sec for 20 clients. This increase, in part, is due to the I/O intensive nature of the update process and can also be attributed to the quad-core machine hosting the CORBA server.

Figure 7 shows the average throughput observed at the client end. As expected, the throughput decreases with an increase in the number of clients updating the CORBA server. The client side throughput decreases from 8000 updates/sec for 1 client to 1600 updates/sec for 50 clients.

### 6.2 Comparison between AMS and AMSB

The next experiment was conducted to evaluate and compare the performance of AMS and AMSB protocols. The experiment was conducted with the protocols configured for 2 replicas and the writes/(reads+writes) fraction was set to 0.5. Results are shown in Figure 8. First, as expected the throughput delivered by the two approaches is lower than the throughput achieved in our baseline experiment. Second, the throughput delivered by the AMS protocol is better than the AMSB protocol for lesser number of clients, while its the other way around when the number of clients is increased. This behavior of the two protocols can be attributed to their implementations. The AMSB protocol makes use of a buffer to hold and block updates until the previous operation is complete. The maximum size of this buffer is equal to the number of clients, and the efficiency of this protocol is directly related to this buffer size, causing the protocol to achieve a much better throughput for a higher number of clients. For a lower number of clients, the overheads caused by the buffer mechanism of the AMSB protocol result in a reduced throughput as compared to the AMS protocol.

### 6.3 Comparison between AMSB and Causal

In Figure 9, we show a comparison between the performance of AMSB and the Causal protocol. The number of replicas was again set to 2, and the writes/(reads+writes) fraction was set to 0.5. The results clearly indicate that a better scalability can be achieved by using the causal protocol. While the AMSB protocol seems to have saturated after achieving a throughout of around 40,000 updates/sec for 25 clients, the throughput for Causal protocol was around 115,000 updates/sec and steadily increasing for the same number of clients. This behavior is caused by the fact that the clients can only contact the master server for the AMSB protocol, while for the Causal protocol either of the replicas can be used.

### 6.4 Performance of the Partitioning protocol

In this experiment, reported in Figure 10, we made use of the partitioning protocol to partition a large list into 3 and 6 partitions for two different experiments. We measured the average time taken

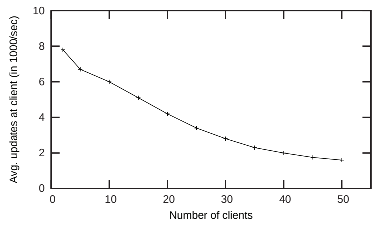


**Figure 7: Variation of update rate as seen by a single client when using a CORBA based single server implementation**

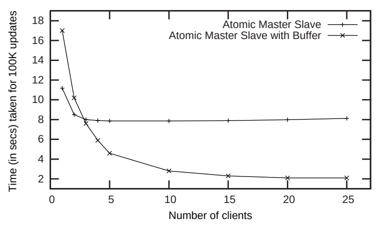


**Figure 8: Variation of time taken for 100K updates with number of clients using AMS and AMSB protocol**

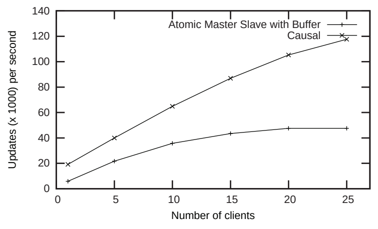


**Figure 9: Variation of update rate with number of clients using AMSB and Causal protocol**

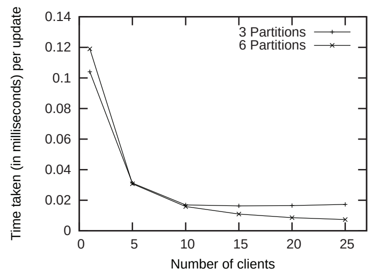


**Figure 10: Affect of the number of partitions for a shared List when using a Partitioning protocol**

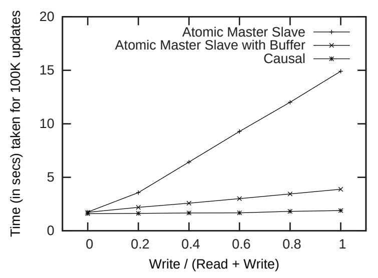


**Figure 11: Variation of time taken for 100K updates with changing the fraction of writes**

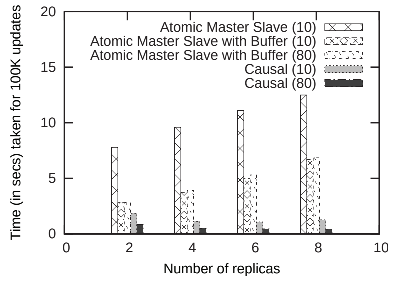


**Figure 12: Variation of time taken for 100K updates with the number of replicas for various protocols**

for each update to the list for varying number of clients. We observe that as the number of clients increases the difference in time-taken for each update increases. Clearly, a higher number of partitions leads to better performance when the number of clients is high.

### 6.5 Effect of fraction of writes on throughput

This experiment was conducted to study the effect of increasing the fraction of writes, i.e. Number of Writes / (Number of Writes + Number of Reads) on the performance of various protocols. As shown in Figure 11, the throughput achieved by the Causal protocol has almost no impact because of the increasing fraction of writes. This is because under the causal protocol the replicas synchronize with each other at a constant frequency and therefore only the most recent write is propagated to other replicas. The number of writes is essentially determined by the synchronization frequency. The AMSB protocol is also able to reduce the impact of increasing fraction of writes because of the buffer that it utilizes. Finally, the impact of increasing fraction of writes is very evident for the AMS protocol. The AMS protocol does nothing to stop the propagation of writes to the replicas, and this contributes to the steep decrease in throughput.

### 6.6 Effect of replicas on throughput

In this experiments we study the effect of the number of replicas on the throughput achieved by various protocols. The results are shown in Figure 12, and the numbers in parentheses indicate the number clients used for that experiment. As expected, the throughput achieved by atomic protocol implementations decreases with an increase in the degree of replication. The AMS protocol suffers the worst degradation in performance because it does nothing to reduce the number of writes that are propagated to the replicas. The Causal protocol on the other hand witnesses an increase in throughput with an increase in the number of replicas. This is because the clients can connect to any replica in case of the Causal protocol.

## 7. RELATED WORK

*Message Passing Programming Model.* Message passing programming model provides developers a way to harness the collective computational capability of a cluster of workstations that do not share physical memory. Message passing is pre-dominant programming model for parallel computing on a cluster of workstations. MPI or the Message Passing Interface [16] is a message passing standard that has been developed to standardize the message passing programming model. Amongst the well known libraries for message passing programming model, PVM or the Parallel Virtual Machine [18] library and the TCGMSG [21] library are the more widely used ones. Message passing fully exposes the distributed nature of the memory system to the developer, providing only the primitives to transport data from one node to another. The developer needs to know about the source and the destination of the data, and when a transfer needs to initiated between the two entities. It is generally considered harder to write parallel programs using the message passing model, but because of the application-level semantics that can be exploited by the developer they often times turn out to be more efficient. CAPSULE provides a much higher and user-friendly abstraction for enabling state sharing between System S entities that run on a cluster of workstations. To achieve efficiency CAPSULE makes use of code generation techniques to tailor the generated code to suit the usage scenario provided by the developer and the target runtime.
*Software Distributed Shared Memory.* Software distributed shared memory systems or DSMs use native message passing facilities to implement and provide a shared memory abstraction over a cluster of workstations. The shared memory abstraction makes this programming model easier to use, but because it hides a lot of details from the developer it is unable to exploit any application-level knowledge and is therefore less efficient than the message passing model. TreadMarks [4] is one of the better known realizations of this programming model. Munin [5] is a DSM that implements

several schemes of memory coherence, which are dependent on the object type declared by the programmer. Midway [6] is another DSM that supports for multiple consistency models to be used in the same program, but the model needs to be explicitly attached with the shared object. CAPSULE makes use of code generation to embed custom protocols into the shared object, and this is done automatically based on the user inputs and the runtime information. CAPSULE is also distinguished by its use of SVDL to specify the deployment time composition of the shared variables.

More recently, Sinfonia [2] proposes a new paradigm for building scalable distributed systems. It provides efficient and consistent access to shared memory. The memory is exposed as a linear address space and makes use of a novel mini-transaction primitive. CAPSULE and Sinfonia are targeted at different usage scenarios; while Sinfonia focuses on providing a consistent view of the data, CAPSULE focuses on providing a range of consistency models that may fit the needs of different applications.

*Distributed Objects, Object Stores and Services.* Distributed objects, like the ones proposed in [17] provide an easy to use interface to distributed shared state, and provide ways to use multiple consistency models. However, such frameworks do not make use of code-generation and therefore lack the performance, and lack the customizations that are possible using SVDL.

Modern day web-applications pose problems that are associated with the management of large amounts of data. To get access data at this scale researchers have proposed large distributed object stores [12, 8]. These object stores offer rapid access to generic stored objects, reliability and handle large number of objects. In comparison, CAPSULE offers access to a small number of specialized shared data objects, here the objects are specialized to offer high-performance and scalable access to shared data.

More recently, institutions have been developing services to help build large distributed systems. These, amongst others, include services like PNUTS [9], GFS [14], MapReduce [11] and Chubby [7]. These systems attempt to offer difficult to implement functionalities as a service and multiple applications can rely on such a service. CAPSULE can be considered as a service for enabling state sharing, but is targeted at a very different domain as compared to the above-mentioned related work.

Some systems make use of databases and filesystems to enable state sharing. However, such mechanisms fail to exploit the weaker consistency guarantees, and offer customizations and performance that are expected by the streaming applications. Memcached [15] is distributed memory object caching mechanism that is intended to be used with databases to speed up the web-application by caching the results from recent queries. Again, CAPSULE offers specialized objects that offer efficient access to shared data, the focus is on the optimizing the shared object implementation to speed-up the access to data that resides in the behind the shared object facade.

## 8. CONCLUSIONS & FUTURE WORK

In this paper we described CAPSULE which is a code-generation and runtime framework for enabling state sharing between System S entities. Enabling state sharing in a distributed stream processing system like System S poses a very distinct set of challenges, and there are opportunities that can be exploited to meet the challenges. CAPSULE by making use of its code-generation and runtime capabilities is able to customize a shared variable to suit the needs of the usage scenario and the capabilities of the target runtime. CAPSULE is still under development and even coming up with a full-fledged set of configuration parameters for a shared variable and using them to arrive at the shared variable customization is going to be an interesting challenge in the future. We are also exploring the idea of pre-customized shared variable for certain scenarios which are encountered very often by the users of System S. We would also like to explore the possibility of using shared variables as a way of parallelizing certain stream processing operators.

## 9. REFERENCES

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

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, 2007.

[3] L. Amini et al. SPC: a distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, 2006.

[4] C. Amza et al. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29, 1996.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. *SIGPLAN Not.*, 25(3), 1990.

[6] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. Technical report, Pittsburgh, PA, USA, 1993.

[7] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[9] B. F. Cooper et al. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2), 2008.

[10] The official CORBA standard from the OMG group. `http://www.omg.org/docs/formal/04-03-12.pdf`.

[11] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[12] G. DeCandia et al. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6), 2007.

[13] B. Gedik et al. SPADE: the System S declarative stream processing engine. In *SIGMOD*, 2008.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5).

[15] memcached journal = `http://www.danga.com/memcached/`,.

[16] MPI. `http://www.mcs.anl.gov/research/projects/mpi/`.

[17] K. Ostrowski, K. Birman, D. Dolev, and J. H. Ahnn. Programming with live distributed objects. In *ECOOP*, 2008.

[18] PVM: Parallel virtual machine. `http://www.csm.ornl.gov/pvm/`.

[19] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, pages 507–528, 2010.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[21] TCGMSG message passing library. `http://www.emsl.pnl.gov/docs/parsoft/tcgmsg/tcgmsg.html`.