CLASP: *Col*laborating, *A*utonomous *S*tream *P*rocessing Systems

Michael Branson[1], Fred Douglis[2], Brad Fawcett[1], Zhen Liu[2], Anton Riabov[2], and Fan Ye[2]

[1] IBM Systems and Technology Group, Rochester, MN USA
[2] IBM T.J. Watson Research Center, Hawthorne, NY USA

Abstract. There are currently a number of streaming data analysis systems in research or commercial operation. These systems are generally large-scale distributed systems, but each system operates in isolation, under the control of one administrative authority. We are developing middleware that permits autonomous or semi-autonomous streaming analysis systems (called "sites") to interoperate, providing them opportunities for data access, performance improvements, and reliability far exceeding that available in a single system. Unique characteristics of our system include an architecture for the management of multiple cooperation paradigms depending on the degree of trust and dependencies among the participating sites; a multisite planner that converts user-specified declarative queries into specifications of distributed jobs; and a mechanism for automatic recovery of site failures by redispatching failed pieces of a distributed job. We evaluate our architecture via experiments on a running prototype, and the results demonstrate the advantages of multi-site cooperation: collaborative jobs that share resources, even across only a few sites, can produce results 50% faster than independent execution, and jobs on failed sites can be recovered within a few seconds.
Keywords: System S, streaming data analysis, Grid computing, Virtual Organizations, planning.

1 Introduction

Data stream processing systems take continuous streams of input data, process that data in certain ways, and produce ongoing results. There are currently a number of data stream processing systems in research [1–4] or commercial [5] operation. These systems are generally large-scale distributed systems, but each system operates in isolation, under the control of one administrative authority. Generally speaking, data that are brought into one such system are available to any application running on the system, and similarly any data created by one application are immediately available to other applications. This sharing is conducive to improving performance and scalability through the synergy of overlapping queries within one system [4, 6]. However, the scale and functionality of an individual system can still be limited when facing extreme data rates (e.g., telemetry from radio telescopes [7]) or complex environments (e.g., supporting

real-time disaster response). Additionally, resources (such as input data streams) that are available to one system are inaccessible to other systems.

In this paper we describe a middleware for **Co**ll**a**borating, **A**utonomous **S**tream **P**rocessing systems (**CLASP**). It sits above separate data stream processing systems and enables these systems to cooperate. We assume that each system, which we call a *site* in the larger cooperative environment, is at least partly autonomous. Thus the extent to which different sites cooperate is a matter of policy, determined by the administrators of each of the sites involved.

CLASP allows sites to benefit in several respects. They can share data sources that were owned and available individually. Thus a site can access a much wider spectrum of data input, greatly increasing the breadth of its analysis. They can share *derived* streams, which are processed results of existing applications, thus avoiding duplicating processing done by other sites and improving efficiency. They can help each other absorb any sudden increase in workload or decrease in resources by rebalancing processing across sites. They can also improve the reliability of job execution by recovering jobs from failed sites.

The middleware has been designed and prototyped in the context of System S [8], a project within IBM Research to enable sophisticated stream processing using arbitrary application logic (rather than relational algebra operations such as used in several other streaming analysis systems [1–3]). Although some details like application interfaces are specific to System S, the architecture itself is generic enough for the interoperation of streaming systems of other kinds.

We make several contributions in this paper. We analyze what functions are needed for stream processing sites to collaborate and propose an architecture that provides them. We extend the traditional Virtual Organization [9] (VO) concept to allow sites to form different VO structures based on the degree of mutual trust and coordination. We implement the architecture on a representative streaming system (System S) to demonstrate its feasibility and evaluate the benefits sites can gain through real testbeds and applications.

The rest of the paper is organized as follows. The next section describes System S in greater detail. Policies governing site interaction follow in Section 3. The architecture of **CLASP** is described in Section 4. Section 5 reports experimental results using a real testbed and application. The paper finishes with related work and conclusions.

2 System S

The goal of System S is to extract important information from voluminous amounts of unstructured and *mostly* irrelevant data. Example applications of such a system include analyzing financial markets (predicting stock value by processing streams of real-world events) [5], detecting patterns of fraudulent insurance claims, supporting responses to disasters such as Hurricane Katrina (based on vehicle movement, available supplies and recovery operations), or processing sensor data such as telemetry from radio telescopes [7] or volcanic activity [10].

We summarize the architecture of System S as a representative of streaming systems and describe some of its key components:

User Interface (UI) Users pose *inquiries* to the system through a front end to answer certain high-level queries. For example, "Show me where all bottled water is in the hurricane area." After the raw data have been processed by application logic (e.g., filtered, joined, and analyzed), results are passed back to the UI via data streams, where they can be presented to the user for further exploitation.

Inquiry Service (INQ) accepts specifications of the desired final results in a format called Inquiry Specification Language (ISL), which depicts the semantic meaning of the final results and specifies user preferences such as which data sources to include or exclude [11]. Given an inquiry, a *Planner* subcomponent [12] automatically composes data sources and processing in the form of jobs to produce desired results. It then submits such jobs to the Job Management component for execution.

Job Management (JMN) A job in System S is a set of interconnected *Processing Elements* (PEs), which process incoming stream objects to produce outgoing stream objects that are routed to the appropriate PE or storage. The PEs can perform stateless transformation or much more complicated stateful processing. System S reuses PEs among different applications when possible to avoid redundant processing.

Stream Processing Core (SPC) manages the execution of PEs [13, 8]. It supports the transport of streams consisting of Stream Data Objects between PEs and into persistent storage. It also provides adaptive connectivity and fine-grained scheduling of communicating applications.

With the exception of INQ, these components map reasonably closely to other data stream analysis systems and are used here as a representative example. INQ is, by comparison, unique to System S: other systems do not have such automatic application composition capability and jobs are usually hand-crafted.

Each System S site runs an instance of each of these system components, possibly as a distributed and fault-tolerant service [14]. Each site may belong to and be managed by a distinct organization; administrators who manage one site generally have no control over another site. Collaboration among multiple sites is thus similar to Grid Computing [9]: sites share resources but retain substantial local autonomy.

As with the Grid, sites that want to collaborate for common goals and benefits can negotiate and form *Virtual Organizations* (VOs) [9]. However, there exist unique requirements in the streaming context, including the need for higher degrees of scalability and various administrative relationships among sites. Section 3 describes how we address these issues.

3 Virtual Organizations and Common Interest Policies

Sites that want to collaborate can form VOs. The members of a VO formalize their permissible interoperations as a *Common Interest Policy* (CIP), which

specifies how they may share various types of resources and processing. VOs can be either *Federated* or *Cooperative*.

A Federated VO has an appointed leader site that assumes a coordination role and is able to exert a level of control over the other sites. This VO is appropriate when the sites share a common set of goal(s) that they want to achieve, or are all subject to a common authority. It allows the VO Lead to optimize resource and processing usage for the common good of the VO.

In a Cooperative VO, there is no central point of authority. VO members interact as peers of each other; they are independent and may have separate agendas. They may interoperate out of altruism, giving access to some resources freely, or they may charge a cost for access (cost could be monetary or credits in some sort of virtual economy).

VOs may have different relationships with each other. A whole VO can be included hierarchically as a member of another larger VO [15, 16]. This allows sites to scale up for wide scope of collaboration. Two or more VOs may have common members which belong to these VOs simultaneously. The kinds of resources the common members share within each of these overlapping VOs, however, can be completely different. The exact resource sharing within the VO is specified by its CIP terms.

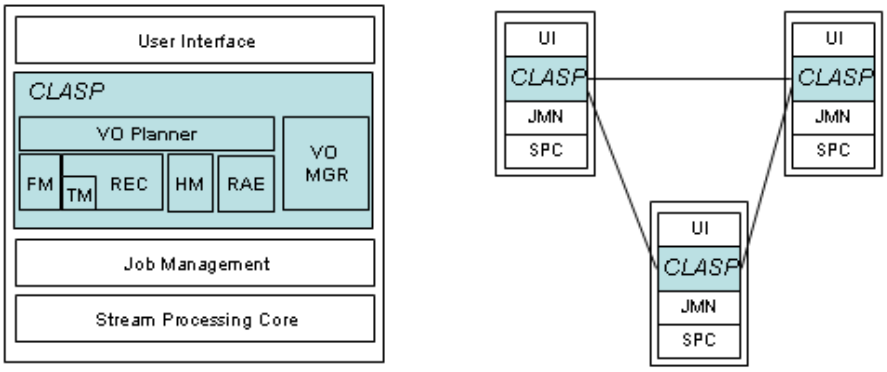
3.1 CIP Terms and Agreements

A CIP contains terms that dictate resource sharing, such as:

- Which set of data streams and locally stored data can be shared by which other remote sites. The set can be defined based on attributes such as the data type or data rate.
- Which set of processing resources can be used to run jobs from which other sites; which kinds of PEs coming from which other sites will the local site execute.
- In times of failure, which sites will perform what function (e.g., monitor, backup data, recover jobs) of the failure recovery process.

The CIP is known by all the members in a VO. By specifying these terms, VO members advertise resources that others may request to use. However, it does not guarantee access, since multiple members may request a resource that can only be used exclusively. Therefore, a VO member must reserve a resource in advance by establishing an *agreement* with the providing member to secure access to the resource for some duration.

Besides defining the kinds of resource sharing that are possible in a VO, the CIP also specifies what parameters are associated with an agreement (such as quality of service levels, costs, and limitations on the resource usage). Once established, this agreement must then be referenced when accessing this resource. The agreement's terms and conditions, along with costs and penalties, will be continuously monitored by some auditing functions at both System S sites providing and consuming the resource.



(a) CLASP architecture (b) Multiple collaborating sites

Fig. 1: CLASP architecture includes several components. They provide the functions needed for sites to collaborate.

This notion of agreement shares similarities with the WS-Agreement specification from the Grid community [17]. For System S, a portion of a CIP term serves as the analogy to the WS-Agreement Agreement Factory and provides the creation template that is needed for creating an agreement between the provider and consumer of the resource. More details are presented in Section 4.2.

4 Architecture

4.1 Overview

Figure 1(a) shows the detailed CLASP architecture on one site. UI, JMN and SPC are single-site components and CLASP is between the UI and JMN. Multiple sites can work together through the interaction of their CLASP middleware (illustrated in Figure 1(b)). CLASP has a number of components providing various functions to support collaboration.

VO Manager deals with the construction of VOs and decisions on permissible cross-site resource usage; Section 4.2 provides details.

VO Planner produces plans utilizing resources from within the VO and partitions a global plan into a distributed job containing multiple subjobs. It is described further in Section 4.3.

Resource Awareness Engine (RAE) provides information about available resources to the VO Planner; see Section 4.4.

Remote Execution Coordinator (REC) extends JMN to the multi-site case by deploying distributed jobs submitted by the VO Planner. Each subjob in a distributed job may run on a different site (elaborated upon in Section 4.5).

Tunneling Manager (TM) manages tunnels that transmit streams from PEs on one site to PEs on another site (details in Section 4.5).

VO Failover Management (FM) handles site monitoring, arrangement of backup sites, and recovery of jobs after site failures. Failover is discussed elsewhere [18] and summarized in Section 4.6.

VO Heterogeneity Management (HM) is intended to manage the mapping or translation of data types, database schemas, security and privacy labels, and similar features between sites; see Section 4.7 for a brief discussion.

4.2 VO Management

The CLASP prototype supports the formation and management of VOs by using text-based CIP definition files. Each VO has a corresponding CIP file, containing three types of terms: VO type, membership, and sharing. Every CIP file must indicate whether the VO is federated or cooperative. For every member of the VO, there must be a membership term, specifying either a site member or a VO member. The CIP file may contain numerous sharing terms. Each sharing term defines what resources can be shared between which two sites, with attributes and their values, agreement creation parameters (separated by semicolons). Below is an example sharing term:

```
2;2;siteA;siteB;MONITOR_SITE_FOR_FAILURE;SHOULD;
COST:10:INITIATION_COST:100;
SITE_TO_MONITOR:MANDATORY:MIN_MONITORING_FREQ:OPTIONAL:
ACTION_UPON_FAILURE:MANDATORY
```

This term has a type (2, resource sharing), an index (2) of this term among those of the same type, identifiers of the sites involved (provider is siteA and consumer is siteB), what resource is being shared (site monitoring capability), access advice (SHOULD), attributes such as cost and initiation cost (10, 100), and what parameters are available when the term is used as an template to create an agreement, including which parameters are mandatory (e.g. action upon failure) or optional (e.g minimum monitoring frequency). We are currently moving to XML, which will provide a more structured framework for this specification.

We expect human administrators to negotiate and install CIP terms on their sites. To create a VO, one site's VO management component parses the CIP file and contacts other sites' VO management components about the creation of the new VO. When there are hierarchical VO members, all descendants of VO members are notified recursively about the new VO. Once a VO is in place, components can establish agreements according to the CIP terms. A component (such as the Failover Manager) does this by first querying its local VO Management for the set of candidate CIP terms that are applicable to its requirements.

For example, if it needs to find possible providers in a VO to monitor a particular site, it submits a query specifying this capability. VO Management will then search and return the matching CIP terms within the specified VO. The FM component will then analyze the terms and conditions of the returned candidate CIP terms and select the "best" one, e.g. a site that can monitor at a

small cost. After filling in the creation parameters such as monitoring frequency, it calls local VO Management, which will in turn contact the VO Management on the provider site to establish the agreement. That VO Management component must contact the providing component and gain its commitment to support the agreement. Once established, the agreement will be referenced when making the inter-site request. The agreement is terminated after its lifetime, or explicitly by the requester.

4.3 VO Planner

The VO Planner is unique to System S. It automatically produces plans that utilize data sources and PEs from all sites in the VO. It accepts inquiries that describe the semantics of desired final results in Inquiry Specification Language (ISL) [11]. The Planner reads in the semantic description of data sources and the required input and output streams of PEs, and uses a branch and bound search algorithm [12] to find plans that can produce the final results.

Given one inquiry, the Planner produces multiple distributed plans in the form of flow graphs, consisting of interconnected PEs and data sources. These plans have different performance/cost tradeoffs and can be presented to the user, who can decide which one to deploy. The planner then partitions the selected plan into multiple sub-plans, each of which is a subjob assigned to one member site for execution. The planner also inserts tunneling PEs into subjobs; each pair of sink and source tunneling PEs transport one stream across sites. Finally, a distributed job that contains multiple subjobs, each of which contains a normal job (for data processing) and multiple tunneling PE jobs (for data transportation), is produced and submitted for execution.

Plan composition within the VO Planner is implemented using a plan solver module that operates on an abstract formulation expressed in Stream Processing Planning Language (SPPL) [12]. SPPL is designed to enable efficient planning in stream processing by introducing language primitives that natively model streams. The semantics of data sources and PEs are represented using OWL ontology [19] files. Since the semantic descriptions are relatively static, these files do not change frequently. When a site joins a VO, it can copy these files over to the VO Planner's site.

4.4 Resource Awareness Engine

Resource awareness refers to the propagation of information about data sources, PEs, and other kinds of resources among multiple collaborating sites. Sites need such remote resource information for operations such as planning, failure recovery. Such information may be stored in relational or semantic data stores, shared memory, or text files. The component that facilitates information propagation among sites is the *Resource Awareness Engine* (RAE).

We intend to use ROADS [20], a resource discovery service, as the basis for this component. ROADS allows multiple sites to query and search for resource information from others. The RAE components on these sites will form

a tree hierarchy, whose exact topology depends on the trust and administrative relationships among sites. Each site's RAE will publish its resource information in a highly condensed summary format. The summaries from child sites will be aggregated by a site's RAE and propagated further up the tree. Thus each RAE will have the aggregated summary about the resource information of all its descendants, and the root RAE obtains the summary of all resource information.

When a site needs to query resource information, it sends a query to the root RAE. The root will evaluate the query against the summaries of its child branches, and find out which branches have the required resource information. It will forward the query down these branches. Each RAE in the hierarchy will follow the same process. Finally the RAEs possessing matching resource information will return it to the requesting site. The details about how summaries are produced and queries are evaluated against them can be found in [20].

For the prototype described in this paper, the RAE is integrated directly with the VO Planner. That is, the Planner is given a configuration file with the description of data sources in each site in its VO. Then as it generates new distributed jobs, the Planner augments its view of available derived streams to include the newly created streams on each site, which it can reuse when needed.

4.5 Distributed Execution

The Remote Execution Coordinator (REC) is responsible for the execution of distributed jobs. The VO planner submits a distributed job to the REC of the *owner* site, which is the one from which the inquiry was received. This REC will coordinate the execution of the subjobs, including their recovery upon failures. The REC dispatches the subjobs to the RECs on the corresponding execution sites, as specified by the planner. An example is illustrated in Figure 2. Site 3 is the owner site and its REC executes the third subjob and dispatches two other subjobs to Sites 1 and 2 for execution. The REC at the owner site maintains a subjob table about which subjobs are running at which other site. The table is used for recovery of subjobs on failed sites.

The REC executing a subjob first parses its Job Description Language (JDL) to identify one normal job, and multiple tunneling PE jobs. One thread is launched to handle each of them. The thread customizes the JDL, such as assigning a host for each PE. Then it deploys the job through its local Job Management. For a source PE job, the REC needs to contact the local Tunneling Manager responsible for assigning the network address and port on which the source PE will be listening for incoming connections. It deploys the source PE job and reports the assigned network location to the REC at the owner site. For a sink PE job, the REC needs to query the REC of the owner site for the network location of the corresponding source PE. Then it configures and deploys the sink PE job.

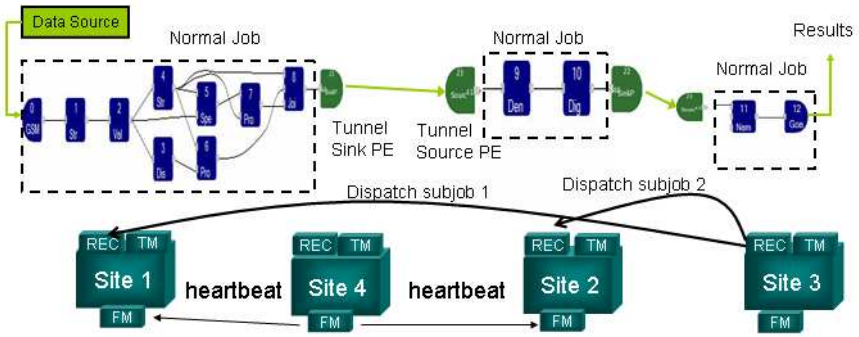


Fig. 2: Execution of a distributed job consisting 3 subjobs. Owner Site 3 executes one subjob, and dispatches two subjobs to Site 1 and 2 for execution. Site 4 monitors Sites 1 and 2.

4.6 Failure Recovery

Failover in **CLASP** has been described elsewhere [18], with emphasis on the problem of identifying which sites are most appropriate for failure recovery in a large-scale VO with many available alternatives. Here we describe the implementation for detecting and handling failures.

The FM at the owner site arranges the failover monitoring for sites executing subjobs. By querying CIP terms, it finds which sites can monitor the liveness of execution sites, using periodic heartbeat messages. When an execution site fails, the FM at a monitoring site detects the failure and notifies the owner site. The REC at the owner site examines the subjob table and finds out which subjobs were running on the failed site. It then dispatches these subjobs to a new execution site, selected from candidate sites returned from VO Management. Algorithms by Rong, et al. [18] can be used for the selection. The new execution site will deploy the subjob.

Although executing normal jobs is straightforward, re-establishing broken tunnels needs special attention. To recover tunnel sink jobs, the REC at the new execution site queries the network location for corresponding tunnel source jobs, then configures and executes the tunnel sink job. The recovery of tunnel source jobs is a bit complex, as the old tunnel sink job might still be sending data to the failed site. The REC deploys such jobs and notifies the owner site about the new network location. The FM at the owner site will inform other execution sites to terminate tunnel sink jobs that send streams to the failed site. These tunnel sink jobs will be restarted using the new network locations of recovered tunnel source jobs. During the above process, new agreements might be created for additional monitoring and execution.

We also envision recovering critical applications from failed sites, even when they run entirely within the site that fails. This will require advance registration

of the jobs to resubmit, with an agreement with another site to monitor the site making the request and to restart the critical applications if needed.

4.7 Heterogeneity

Our current prototype assumes a homogeneous environment. In the more general case, each site may have differences in its operating environment. This heterogeneity can arise in the runtime environment, type system, security and privacy policies, user namespace, and other aspects.

The general approach to heterogeneity is through *mapping functions* and common base agreements. The CIPs that govern how sites interoperate must specify operations to perform to ensure consistency. Differences in data types will be handled through explicit conversion functions: for example, converting a nine-digit US ZIP code into a five-digit one would involve truncating the additional level of detail. For security, System S assumes lattice-based [21] secrecy and integrity policy models [22]. Each site will understand the format and implied relationships of security labels used by all sites; the access rights and restrictions encoded within a security label are uniformly applicable throughout all the sites. We will address operation in heterogeneous environments in the future.

5 Experimentation

5.1 Test Environment

We have implemented the **CLASP** architecture in Java (with the exception of the tunneling PEs, written in C++). The prototype currently has about 40,000 lines of code. We use a testbed that consists of Linux SUSE 9 machines. Each machine has 2 Xeon 3.06 GHz CPUs, 800MHz, 512KB L2 cache, 4G memory and 80G Hard drive. They are connected through a 1Gbps LAN. Multiple machines can be grouped together as a System S site, which **CLASP** runs above. For most experiments, we use a Federated VO that contains four sites, one of which is a backup site, while the others are execution sites.

The goal of experiments is two-fold. 1) Quantify the benefits collaborating sites can gain compared to operating individually. We use the total number of produced results as the main metric. 2) Benchmark the time overhead of basic operations of CLASP, such as planning, job submission, and failure recovery. This gives us a basic understanding of the efficiency of the system.

To evaluate our system, we use an application we entitle "Enterprise Global Service" (EGS). EGS is intended for enterprises to monitor the quality of service of their customer service personnel. Customers talk with service representatives through a corporate VoIP network. A business analyst can issue various inquiries to examine the status of employee services. These inquiries include: find the location and "courtesy level" of a particular employee, find the satisfaction level of a particular customer, etc. We use a VoIP traffic generator [23] to produce the VoIP streams between employees and customers. Each inquiry's job contains about 15 PEs and a job produces results continuously during its lifetime.

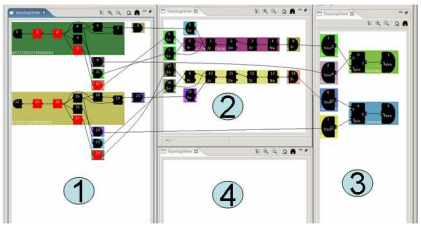


Fig. 3: Two distributed jobs are deployed within a VO of 4 sites. Each job has 3 subjobs that run on Sites 1, 2 and 3. Tunnel PEs connect subjobs across sites.

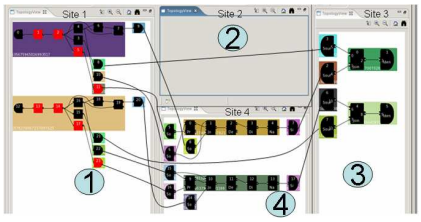


Fig. 4: After Site 2 fails, the two subjobs running on Site 2 are recovered on Site 4. Tunnel PEs are reconnected so that the two distributed jobs continue producing results.

Figure 3 shows an example of two distributed jobs deployed in the VO. Each of the two jobs (`location` of SHIMEI, location of EMILY) has three subjobs, running on Sites 1, 2 and 3. Roughly speaking, these jobs work as follows: A source PE pulls in all streams from the traffic generator. An annotator PE extracts Real Time Protocol fields and turns them into SDO attributes, then a value-based filter PE removes background noise. A speaker detection PE detects the identities of persons; location/courtesy/satisfaction analyzing PEs produce the location, courtesy or satisfaction of persons. Their results are joined and then filtered based on which person the inquiry is looking for. The final results are reported and shown in a GUI.

Among all the PEs, location/courtesy/satisfaction analyzing PEs are the most computing-intensive. Beyond the mimimum processing required to perform the required tasks, the amount of extra processing they perform on each incoming SDO, defined as the *load level*, can be tuned. In the experiments we vary the load level for them to evaluate the system behavior under different computation intensities; zero load level corresponds to normal processing.

Figure 4 shows what happens after Site 2 fails. Site 4 detects the failure and notifies the owning site, Site 3, which recovers the failed subjobs on Site 4. The tunnel PEs are reconfigured such that cross-site data streams reconnect to the same subjobs recovered at the new site.

5.2 Result Production

We measure the performance of our prototype in several respects. We first compare the number of results obtained by collaborating sites in a VO, or using sites individually, under the same inquiry load. We produce three sets of inquiry load. Within each set, there are 6 inquiries submitted to each of the three execution sites in a VO. An individual site uses its own data sources and resources to produce plans and run the jobs. The sharing of streams is confined within each site. When the same 18 inquiries are submitted to the VO, the VO planner produces jobs that can reuse remote derived streams across sites.

Set	Site 1	Site 2	Site 3
Set 1	loc SHIMEI	loc SHIMEI	loc SHIMEI
maximum	loc FAYE	loc FAYE	loc FAYE
reuse	cor SHIMEI	cor SHIMEI	cor SHIMEI
	cor FAYE	cor FAYE	cor FAYE
	sat SHIMEI	sat SHIMEI	sat SHIMEI
	sat FAYE	sat FAYE	sat FAYE
Set 2	loc SHIMEI	cor LEONARD	sat SHIMEI
minimum	loc FAYE	cor NORMAN	sat LEONARD
reuse	loc ENRIQUE	cor MARK	sat MARK
	loc NAOMI	cor FAYE	sat EMILY
	loc LEONARD	cor ENRIQUE	sat NAOMI
	loc EMILY	cor SHIMEI	sat FAYE
Set 3	loc ENRIQUE	cor FAYE	sat FAYE
average	sat EMILY	sat NORMAN	cor MARK
reuse	cor NAOMI	sat FAYE	sat LEONARD
	sat NORMAN	cor EMILY	sat NORMAN
	loc MARCIA	cor NORMAN	loc SHIMEI
	sat SHIMEI	loc MARCIA	cor LEONARD

Table 1: The 3 sets of inquiries used in the experiments. **loc** refers to getting the location of an employee; **cor** obtains their courtesy; and **sat** computes customer satisfaction.

Due to the sharing of more common processing, jobs running in a VO will generally produce results more efficiently. The more common processing across sites, the higher the savings by sharing existing processing. The three sets of inquiries correspond to different degrees of sharing (shown in Table 1). In the first set, the 6 inquiries (2 location, 2 courtesy, 2 satisfaction) submitted to each site are the same. When a new instance of the same inquiry is submitted, only additional tunneling and result reporting PEs are needed. They correspond to the maximum degree of cross-site sharing.

In the second case, each site has a distinct set: Site 1 has only location inquiries, Site 2 only courtesy inquiries, and Site 3 only satisfaction inquiries. This corresponds the minimum degree of sharing. Inquiries of different sites can share only a few PEs such as the source PE and background noise reduction PE. They have to do the most computing-intensive processing (finding location/courtesy/satisfaction) by themselves. The third set is a middle ground between the two. Each site has a random mixture of inquiries, including different types and person names. The degree of sharing is less than the first but greater than the second set. This is likely what would happen in reality. For each set, we vary the computation intensity of jobs by changing the load level. We let jobs run for 2 minutes, and average the results over five runs.

Figure 5 compares the total number of results of all the 18 jobs in set 1 when running in the VO or individually. They produce about the same amount when the load level is zero. As the load level increases, running in the VO can produce as much as 50% more results, because jobs can tap into the processed results

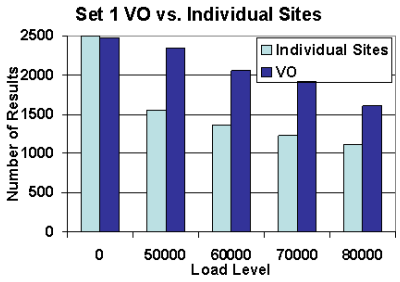


Fig. 5: The total number of results produced by the 18 jobs in set 1, running at individual sites or within the VO.

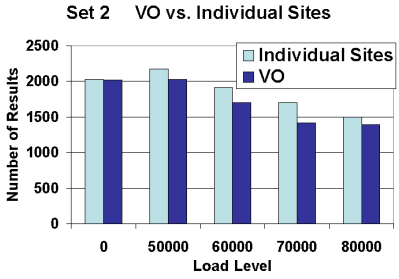


Fig. 6: The total number of results of all the 18 jobs for Set 2, running at individual sites or within the VO.

across sites and avoid duplicating common processing. For those running at an individual site, however, they can only tap into processing within the same site. We also examined the number of results produced by each individual job, when running in a VO or one site. The phenomena is similar and we do not elaborate due to space limitations.

Figure 6 compares the number of results for set 2. Jobs running in a VO produce slightly fewer results than in set 1. The reason is that the cost paid for sharing offsets the benefits. In set 2, each site has only one type of job (location, courtesy or satisfaction). Jobs at different sites do not share computation-intensive processing. Thus running in a VO does not reduce the amount of processing much.

On the other hand, there is a cost to pay for a VO. Extra tunneling PEs are one factor. Another is a synchronization effect. A PE consuming SDOs slowly may cause its producing PE to wait since reliable transport is used to send SDOs between PEs. Other consuming PEs receiving SDOs from the same producing PE will have to wait as well. Thus one job that runs more slowly affects other jobs when they share input streams. Set 2 is the worst case where little processing can be shared across sites, thus the savings are not enough to cover the cost.

Figure 7 shows the comparison for set 3. The result is quite similar to that of set 1: running in a VO produces more results. This similarity is because each site has a random sequence of jobs that contains all different types and person names. The common processing across different sites is significant. The VO allows jobs to reuse the processing across site, thus producing results more efficiently.

5.3 Failover

Sites in a VO can backup and recover jobs for each other when some of them fail. Figure 8 shows the details about one VO job's result sequence number change for failover. Around time 402.5s a site fails, then after another 13s the failure is detected. (The detection time depends on the heartbeat interval, which can

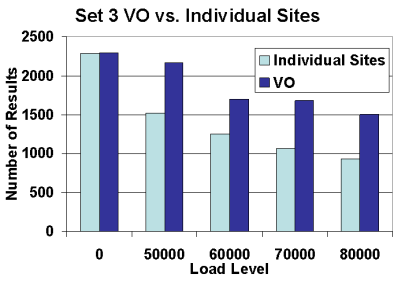


Fig. 7: The total number of results for all the 18 jobs for Set 3, running in VO or individual sites.

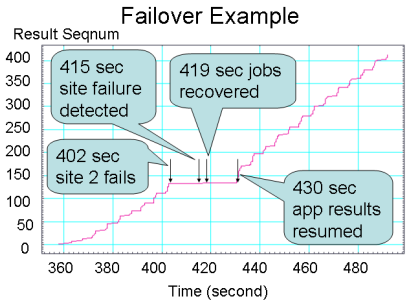


Fig. 8: The job sequence number as a function of time. Once detected, failed jobs are recovered in about 3.5s.

(a) Load level 0.

(b) Load level 80000.

Fig. 9: The aggregate result sequence number for jobs running in a VO and on Site 4 with varying load levels.

be set to achieve the desired detection speed.) In about 3.5s the failed jobs are recovered. Since the job needs some time to rebuild the lost state, it resumes producing results 10s later.

We use the number of results produced to demonstrate the advantage of failover. We run three types of jobs on Sites 1-3 in a VO. Site 4 is monitoring Sites 1-3. Upon the failure of any of them, Site 4 will recover subjobs running on the failed site. Site 4 also runs three types of jobs on itself. We let all jobs run for one minute, then we kill Site 1. After jobs are recovered on Site 4, we let them run for another two minutes.

Figure 9(a) shows the *aggregate sequence number* as a function of time, for all VO jobs and all Site 4 jobs, when the load level is zero. At any time, the aggregate sequence number for a collection of jobs is defined as the total number of results produced by these jobs up to that time. Starting around time 7240s, all jobs are producing results. At around 7300s, Site 1 fails. The aggregate sequence number for VO jobs stays flat, while for Site 4 jobs it is still increasing. After Site

4 detects Site 1's failure and recovers its subjobs (around 7350s), the VO jobs start to produce results. Since the load level is low, there is sufficient processing capacity on Site 4 to accommodate the failed subjobs without affecting those of its own. The speed of sequence number increase for Site 4 after failover remains about the same as before.

Figure 9(b) shows the same comparison under load level 80000. The sequence numbers increase more slowly. Eventually Site 4's and VO jobs produce about 1100/450 results, less than the 1400/650 results when the load level is 0. Although jobs for both Site 4 and VO produce less results, it is still much better than without failover (the VO jobs would not produce any more results).

Another interesting observation is that Site 4's jobs produce results more quickly between 7800-7850s. This is due to the lack of any synchronization effect during recovery. Since all jobs receive input streams from the same data source, more jobs will slow down the producing rate of the data source. When VO jobs have failed but not recovered, only Site 4's jobs are consuming data.

5.4 Planner

Plan Solver Performance We measure the time it takes the VO planner to find plans that produce the desired final results. The Stream Processing Planning Language (SPPL) solver we use to implement the VO planner has been evaluated within one single site. It is scalable with large numbers of PEs, source and plan sizes [12]. The VO planner adds tunneling PEs to plans and optimizes plans for distributed metrics such as minimizing cross-site bandwidth consumption, calculated using bandwidth consumption for PEs and sources that produce cross-site streams.

We run the VO planner on a 3GHz Intel Pentium 4 PC with 4 GB memory. We use a setting that includes 5 sites. Data sources are uniformly randomly assigned to a site and each data source is available at that site only. PEs are available on all sites. This is reasonable because PE code can be easily transfered and installed at other sites (assuming they are secure and trusted). PEs and sources are given randomly constructed descriptions of their inputs and outputs, and random output bandwidth.

Since there could be many PEs that are not relevant to an inquiry, the processing graphs are likely to be of relatively small sizes. However, the planner still takes time to search through plans including irrelevant PEs. To model this scenario, we vary the number of PEs per site from about 72 to 1500, most of which are not relevant to the specified goal. To ensure plans that produce a given final result do exist, we generate random global processing graphs first and use their final results as input to the planner. There exist only 2 candidate plans of 6 nodes each (excluding tunneling PEs) for the goal. We average the results over 10 runs.

The planning time as a function of the number of PEs per site is presented in Table 2. We can see that it takes the planner less than one second to find the optimal plan for sites having up to about 160 PEs. Even in the case of 1500 PEs,

Number of PEs per site	Time to Optimal Plan (s)
72	0.37
102	0.36
162	0.51
312	1.46
612	2.20
1512	8.39

Table 2: Planning times for optimal 6-PE plans, as a function of the number of PEs per site.

Number of PEs	Time to first plan (s)
5	0.17
10	0.592034
20	0.753005
40	0.680179
50	1.01968
100	0.966948

Table 3: Planning times for the first plan, as a function of the total number of PEs in the plan.

it is only a little bit over 8s. Since many streaming jobs are expected to run for a long time, spending a few seconds to find an optimized one is reasonable.

We further evaluate the time to the first plan as a function of the plan size, i.e., the number of PEs in the plan (see Table 3). In all cases the planner can find a reasonably good plan within about a second. In general, the larger the plan size, the greater the time it takes. However, this time is not completely monotonic with the size of the plan, because the search time depends on the structure of individual plans as well.

Since proving optimality is a more difficult problem, it takes long time to decide whether a discovered plan is optimal. However, empirical results show that plans found initially are close to optimal ones. In 10 randomly generated planning problems that require 6 PEs and sources in the plan, the solver considered on average 104 candidate plans. The first plan is found between the first 7.3% and 25% of the plan search time. This plan is within 1.2% of the optimal one, using a quality measure that combines an additive PE and source quality metric and inter-site bandwidth consumption.

Hence, when the search takes longer than several seconds, we terminate the search early and present the current plan for deployment assuming that it is close to the optimal. We leave further improvement on the scalability of the SPPL solver to future work.

Agreements-Driven Replanning A prerequisite to successfully deploy a distributed job is that all agreements are established. To avoid incurring possible costs before job deployment, the planner does not establish agreements at planning time. Instead, agreements are established when the job is being deployed. If not all of the required agreements can be established, one must replan.

We have measured the time that replanning takes in the EGS application by distributing the job to 3 sites and configuring the sites to reject initial agreements. The planner then replans the jobs with higher priority (and possibly higher site-dependent execution budget). Replanning was performed 3 times before deployment, resulting in higher job priorities and different plan partitioning. The whole cycle requires less than 7s. Although in this case replanning happens

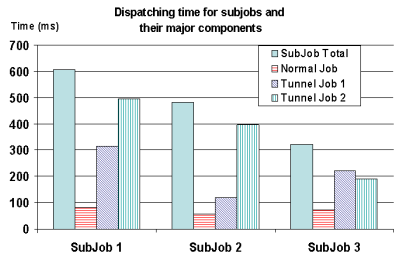


Fig. 10: The dispatching time details of a distributed job, from submission to finally it is deployed and ready to run.

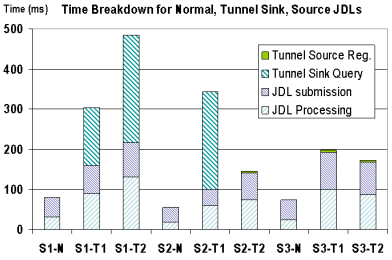


Fig. 11: The detailed breakdown of each subjob. Subjob 1 has two tunnel sink jobs, subjob 2 has one sink and one source, subjob 3 two sources.

completely automatically, the VO planner provides APIs for developing more sophisticated GUIs to allow human feedback when replanning is needed.

5.5 Job Deployment Time

To understand the responsiveness of the system, we also measure the time it takes to deploy a distributed job. This is from the submission of the JDL of a distributed job, to the dispatching of its subjobs to other sites, until finally all subjobs are up and ready to process data. We use the same JDLs as before and they each contain about 20 PEs (including tunneling PEs).

Figure 10 shows the detailed time breakdown for a distributed job, with three subjobs, each of which has two tunnel jobs and one normal job. For each subjob, a separate thread is launched to dispatch it to the corresponding site. Thus the overall time is dominated by the longest subjob. After a site receives a subjob, it processes the JDL first, then it launches one thread for each of the jobs: the normal job and the two tunnel jobs. The time for a subjob is in turn dominated by the job taking the longest time. From Figure 10, subjob 1 takes the most time, about 600ms. The other two subjobs take about 500ms and 400ms, respectively. Within each subjob, one tunnel job takes the longest time. The whole distributed job takes about 700ms.

Figure 11 shows the finer breakdown for each normal job and tunnel job. We find that the tunnel sink query takes the longest time. The reason is that, although a tunnel sink job can be deployed almost simultaneously as its tunnel source end, it has to query and wait for the tunnel source to register the listening IP address and port. Thus a tunnel sink is always deployed later than its source end. We plan to explore a "gateway" approach where multiple cross site streams can be multiplexed between a pair of gateway PEs to further improve the performance.

6 Related Work

CLASP has a strong relationship, yet significant differences, with two general areas of computing: Grid computing [9] and streaming data analysis [3, 2, 1]. With respect to Grid computing, a recent article [24] highlights the similarities between cooperative stream processing and Grid computing. They describe similar environments: "distributed, multidisciplinary, collaborative teams" that attack problems in a distributed fashion due to the nature of their various "intellectual, computational, data, and other resources." Indeed, our system adopts some Grid constructs, such as VOs. In addition, there has been substantial work in matchmaking between different organizations based on required capabilities (e.g., Liu, et al. [25] and the recent work on WS-Agreements [26, 17]).

At the same time, there are a number of important differences. Our architecture supports multiple cooperation paradigms, including Federated and Cooperative (peer-to-peer) VOs. It allows sites to collaborate more closely, with hierarchical layers of VOs to provide arbitrary scalability. This is suitable for complex stream processing that cannot be easily broken into smaller and similar pieces and requires complementary contributions from all sites. The distributed planning component of System S is significantly more sophisticated and flexible than the Grid models.

Borealis [3] is a distributed stream processing analysis system with a number of similarities to System S. It has explicit support for fault tolerance [27] as well as contracts to "sell" load between sites in a federated system [28]. **CLASP**, using System S, differs fundamentally from Borealis and other stream processing systems such as STREAM [1] and TelegraphCQ [2] in a number of aspects. First, although each such system itself can be distributed, there is no support for streaming systems belonging to different administrative authorities to work together. They cannot benefit from the sharing of data streams and processing to improve efficiency, reliability, or the breadth, depth and scale of analysis.

Second, System S supports generic application-specific processing rather than database operations— a more difficult problem due to higher complexity, development costs and times to completion [29]. System S has an Inquiry Specification Language that allows users to specify application declaratively at semantic level. This is very important to allow users focus on application level tasks, rather than deal with the complexity of finding the optimum set and interconnection of data sources and PEs.

7 Conclusions and Future Work

In this paper we have demonstrated that **CLASP**, our middleware for cooperating data stream processing sites, enables such sites to increase the scale, breadth, depth, and reliability of analysis beyond that available within a single site. Experiments with our prototype have demonstrated the performance benefits gained from reusing processing from other sites, as well as quantifying

some of the overhead incurred in the system. There also exist other more qualitative benefits, such as access to remote data sources to broaden the breadth of analysis.

One of the important aspects we will investigate in the future is what mechanisms are needed to support security and trust. The current system works in a benign environment. When sites do not have full trust for each other, or some of them are selfish or even malicious, security checks should be enforced. In addition, as the system evolves, we will incorporate features such as fully dynamic resource awareness and support for heterogeneity.

We also plan to investigate the scalability of the system. Our testbed was a small number of sites, which is probably consistent with typical interoperating agreements one might expect from a system of this sort: in a real system, each site would itself be a very large-scale distributed system. Beyond that, we are currently experimenting with issues regarding large, multilateral agreements, particularly in competitive economic environments in which sites do not provide resources simply out of altruism.

Acknowledgments

We thank the anonymous referees, Lisa Amini, Nagui Halim, Anand Ranganathan, Bill Waller, and the rest of the System S team for helpful feedback on the design of **CLASP** and/or earlier drafts of this paper.

References

1. The STREAM Group: STREAM: The Stanford stream data manager. IEEE Data Engineering Bulletin **26**(1) (2003)
2. Chandrasekaran, S., et al.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: Conference on Innovative Data Systems Research. (2003)
3. Abadi, D.J., et al.: The design of the Borealis stream processing engine. In: CIDR 2005 - Second Biennial Conference on Innovative Data Systems Research. (2005)
4. Pietzuch, P., et al.: Network-aware operator placement for stream-processing systems. Proc. the 22nd International Conference on Data Engineering (ICDE06) (2006)
5. Streambase Systems, Inc.: Streambase. http://www.streambase.com/ (2007)
6. Repantis., T., Gu., X., Kalogeraki, V.: Synergy: Sharing-aware component composition for distributed stream processing systems. In: ACM/IFIP/USENIX 7th International Middleware Conference. (2006) 322–341
7. Risch, T., Koparanova, M., Thide, B.: High-performance GRID Database Manager for Scientific Data. In: Proceedings of 4th Workshop on Distributed Data & Structures (WDAS-2002). (2002)
8. Jain, N., et al.: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: 25th ACM SIGMOD International Conference on Management of Data (SIGMOD 2006). (2006)
9. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: Enabling scalable virtual organizations. Lecture Notes in Computer Science **2150** (2001)

10. Werner-Allen, G., et al.: Deploying a Wireless Sensor Network on an Active Volcano. IEEE Internet Computing **10**(2) (2006) 18–25
11. Bouillet, E., et al.: A semantics-based middleware for utilizing heterogeneous sensor networks. In: Proceedings of the 3rd IEEE International Conference on Distributed Computing in Sensor Systems. (2007) 174–188
12. Riabov, A., Liu, Z.: Scalable planning for distributed stream processing systems. In: Proceedings of ICAPS 2006. (2006)
13. Amini, L., et al.: Adaptive control of extreme-scale stream processing systems. In: Proceedings of ICDCS 2006. (2006)
14. Jacques-Silva, G., et al.: Towards autonomic fault recovery in system-s. In: Proceedings of the 4th IEEE International Conference on Autonomic Computing. (2007)
15. Kim, K., Buyya, R.: Policy-based Resource Allocation in Hierarchical Virtual Organizations for Global Grids. Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'06)-Volume 00 (2006) 36–46
16. Branson, M., et al.: Autonomic operations in cooperative stream processing systems. In: Proceedings of the Second Workshop on Hot Topics in Autonomic Computing. (2007)
17. Andrieux, A., et al.: Web Services Agreement Specification (WS-Agreement), Version 2006/07. GWD-R (Proposed Recommendation), Grid Resource Allocation Agreement Protocol (GRAAP) WGGRAAP-WG (2006)
18. Rong, B., et al.: Failure recovery in cooperative data stream analysis. In: Proceedings of the Second International Conference on Availability, Reliability and Security (ARES 2007), Vienna (2007)
19. Recommendation, W.: Web ontology language (OWL) (2004)
20. Yang, H., et al.: Resource discovery in federated systems with voluntary sharing. in submission (2007)
21. Sandhu, R.: Lattice-based access control models. IEEE Computer (1993)
22. IBM: Security in System S. http://domino.research.ibm.com/comm/research_projects.nsf/pages/system_s_security.index.html (2006)
23. Anderson, K.S., et al.: SWORD: Scalable and flexible workload generator for distributed data processing systems. In: The 37th Winter Simulation Conference. (2006) 2109 – 2116
24. Foster, I.T., Kesselman, C.: Scaling system-level science: Scientific exploration and IT implications. IEEE Computer **39**(11) (2006) 31–39
25. Liu, C., et al.: Design and evaluation of a resource selection framework for grid applications. In: In Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing. (2002)
26. Ludwig, H., Dan, A., Kearney, B.: Cremona: An Architecture and Library for Creation and Monitoring of WS-Agreements. ACM International Conference on Service Oriented Computing (ICSOC'04) (2004)
27. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-Tolerance in the Borealis Distributed Stream Processing System. In: ACM SIGMOD Conf., Baltimore, MD (2005)
28. Balazinska, M., Balakrishnan, H., Stonebraker, M.: Contract-based load management in federated distributed systems. In: Symposium on Network System Design and Implementation. (2004)
29. Stonebraker, M., Çetintemel, U., Zdonik, S.B.: The 8 requirements of real-time stream processing. SIGMOD Record **34**(4) (2005) 42–47