Fakultät für Mathematik und Informatik

Diplomarbeit

Distributed Processing of Data Streams in P2P Networks

Tobias Scholl
Alemannenstr. 11
86199 Augsburg

Erstgutachter:	Prof. Alfons Kemper, Ph. D.
Zweitgutachter:	Prof. Dr. Winfried Hahn
Betreuer:	Dipl.-Inf. Richard Kuntschke

19. September 2005

Prof. A. Kemper, Ph. D.
Lehrstuhl für Informatik III : Datenbanksysteme
Fakultät für Informatik
Technische Universität München

To Nina, Elisabeth, and Hartmut

Abstract

In many industrial and scientific cooperations and communities, globally distributed institutions share their data and computational resources to compose so-called *virtual organizations*. Efficient parallelization, network-aware load-balancing, and distributed processing of heterogeneous data sources are exemplary for arising challenges.

Virtual observatories play an integer role to face data explosion and to support distributed research in the fields of astronomy and astrophysics. The appearance of many general issues in these communities is a great incentive to probe further.

As amounts of data are rising very fast due to technical advances, traditional publish-subscribe approaches such as *data shipping* reach their border of feasibility. Disseminating data as streams to fellow researchers or realizing several key applications such as matching processes over a Grid-based Peer-To-Peer infrastructure could gain significant benefits.

This thesis introduces an approach for distributed data processing which applies concepts from distributed query processing of persistent data on data streams. It describes further an interface to integrate user-defined operations into a data stream management system and demonstrates how such a system can be monitored and evaluated with a prototype.

Contents

List of Figures

Listings

List of Tables

Chapter 1

Data Streams, P2P Computing, and the Grid

Databases as storage for persistent data are a crucial part of many enterprise applications. The sky is the limit for money to be spent on high scalable and performant hard- and software to cope with the many transactions of all-day's business or with the complex decision support queries on a data warehouse.

Many businesses are depending on applications like Customer Relationship Management (CRM) or Enterprise Resource Planning (ERP) and as such, storing and accessing data plays an integer part. State-of-the-art central or distributed databases are used and are potentially reachable all over the world by interface like SQL, ODBC or JDBC. Some might in fact not even use a database due to monetary or historic reasons as they are not in favor of changing running systems and foster gigabyte-scale files by providing access via FTP.

With technologies like Radio Frequency Identifiers (RFIDs), data processing at the stock markets, health care monitoring or network traffic analysis at providers, there are several applications where the processed data can be described better as data streams than persistent data. These data streams are published into a network and users subscribe to these data streams.

How does such a network look like? In these global scale information systems the client-server approach using one or more centralized servers where thousands of clients can attach to is phasing out. The prevalent technology jettisons the idea of a centralized infrastructure in favor to a self-organized systems with autonomous peers, the so-called Peer-To-Peer (P2P) networks.

There are several definitions of what a P2P network is and many emphasize on the dynamic behavior of the individual participants in the network. Every peer joins or leaves the network as it likes. In the context of this diploma thesis, we focus more on the aspect that each participating peer provides the same services although there might be differences in computational power. The peers communicate over an overlay network, meaning that the interconnections between peers are on a logical layer which hides the real physical connections between the

individual peers.

To shape these overlay networks in an efficient manner varies significantly from application contexts and is subject to intensive research. Some exemplary approaches are *structured* P2P networks (using distributed hash-tables), *unstructured* (also known as power-law networks), and *hierarchical* topologies like super-peer networks which make allowances for the fact that not all machines have same characteristics in terms of availability or processing power.

Many of these various systems aim on sharing resources. The kind of resources ranges from physical resources like bandwidth, processing power, and storage to resources on a conceptual level such as content or knowledge.

Sharing resources is also a central concept in grid computing, a discipline in computer science that experiences currently a new revival. A basic principle of grid computing is to join (mostly otherwise idle) resources to solve computational tasks that are too hard for a single machine. Ian Foster et al. extended a grid definition focusing only on hardware and software infrastructure by taking also social and political points of view into account and coined the term of*virtual organizations* [FKT01], which get an increasing importance in several communities. In a different publication [Fos02], Foster proposes a three-point checklist for "Grid-determination":

> A Grid is a system that (1) coordinates resources that are not subject to to centralized control (2) using standard, open, general-purpose protocols and interfaces (3) to deliver nontrivial qualities of service.

1.1 Challenges at the Borders of Disciplines

Several key research areas in the astronomical and astrophysical research communities show the necessity to combine different institutions (and their data archives) in virtual observatories. Two additional driving factors are the explosion in size of the astronomical data sets and the prospect of new research areas. The Internet or Grid-based P2P technologies are exemplary approaches used for the interconnection.

As part of the German Astrophysical Virtual Observatory (GAVO), the Max Planck Institute for Extraterrestrial Physics (MPE) focuses among other research topics on the matching and classification of spectral energy distributions (SEDs). The necessary information is partly distributed over several catalogs which often are situated at different sites. By specifying selection criteria ("objects of interest") and catalogs they are interested in, researchers want to examine the resulting "whole picture". This is provided by joining the different catalogs together and executing user-defined functions on them.

The challenges arising from designing and implementing a stable and efficient architecture providing this functionality are an ideal environment for e-science

cooperations between astrophysics and computer scientists which both sides gain a lot of benefits from.

On the one hand the specific domain knowledge of users' needs and application pitfalls of the former is extended by the expertise of the latter in processing vast amounts of data in database systems or in heterogeneous environments. On the other hand it is a unique opportunity for computer scientists to use a research project on real instead of simulated testing data.

1.2 The StreamGlobe infrastructure

StreamGlobe is a Data Stream Management System (DSMS) for processing and sharing data streams in a P2P environment. Peers can take part in several roles in the StreamGlobe architecture:

Super-Peers are the backbone of the StreamGlobe infrastructure. On these machines all the query processing and data stream routing takes places. They are predominantly powerful stationary servers.

Thin-Peers have the complementary role to the Super-Peers. They publish data streams into the super-peer network and issue subscriptions to those available data streams. They are less powerful and more volatile peers or sensors. Subscriptions on the data streams are written in *Windowed XQuery (WXQuery)* which includes a fragment of XQuery augmented with time-based and element-based windows [KSK05].

StreamGlobe works on XML data streams. There are several mechanism to describe the structure of XML data, XML Schema and Document Type Definitions (DTDs) being the most common. In the following we will concentrate on DTDs as for pragmatic reasons although conceptually XML Schema could be supported as well. XML Schema tends to blow up the description of the data structure and thus may more obfuscate the clarity of the description. Furthermore, handling with XML Schema offers several pitfalls for implementers and so we decided to stick with DTDs.

We use the FluX [KSSS04] query engine for its scalable evaluation of (W)XQueries on streaming data. Using schema information can significantly reduce the buffer consumption during the query execution and therefore can reduce response time (first results come in faster) and memory usage.

The StreamGlobe interface is constructed atop of the Open Grid Services Architecture (OGSA) using its reference implementation, the Globus Toolkit 3. The individual stream processors, process an input stream and transform it into an output stream. The OGSA interface is used for the communication process between the super-peers. Stream dissemination is implemented using a self-defined protocol based on TCP/IP, the StreamGlobe Transfer Protocol (SGTP). Grid

services are special web services that also support inheritance, service properties and stateful (in comparison to stateless) computations.

Many middleware systems currently follow a principle called *data shipping*. When a user subscribes to a data stream, the whole data stream is transmitted to and finally processed at the user's machine. This *conventional approach* puts the burden of processing a query on the subscriber's machine. However, this has major deficiencies. When the same query gets installed at different sites, the transformation steps are redundantly executed. Additionally, data streams are redundantly transmitted through the network and most of the data is filtered at the final peer with projection or selection predicates. Looking at the overall load situation, we see increased peer load (redundant executions) and increased network traffic (redundant and unnecessary transmission).

How can we reduce network traffic and peer load? A first heuristic would perform *shortest path routing*. All query execution is performed at the publisher's peer and therefore all unnecessary data is filtered out before dissemination along the shortest path to the subscriber. Unfortunately, now all the work is done at the publishers peer and we still – remember the same query installed at different peers – have redundant execution.

StreamGlobe overcomes these deficiencies with in-network query processing and multi-query optimization. Query operators are distributed in the network (*query shipping*) and we perform early filtering and aggregation. This is achieved with a *cost-based optimizer* that compares several processing alternatives and chooses the best in terms of a cost function.

This query optimization is conducted by specific super-peers, the *speaker-peers*. To speed up the optimization process, we divide the StreamGlobe network into several subnets and all super-peers of a subnet "elect" a speaker-peer for this subnet. Each super-peer then forwards the meta-data about its queries and streams available to the speaker-peer which in return can use this information for the optimization described above.

We leave it with this overview of the StreamGlobe architecture and provide more details in the individual chapters when there are interfaces to the new components. For further details about the StreamGlobe infrastructure and optimization component, we refer interested readers to the literature [Fen05, Häu05, KSH+04].

1.3 The StarGlobe Interface

The StreamGlobe system has its main focus on P2P Stream Sharing. This thesis describes an extended interface to the StreamGlobe architecture called *StarGlobe*.

A component to distribute the query plans is designed as part of the Grid Service which provides the basic functionality of a peer in the StarGlobe network. A plan is issued to the peer which registered the query and is recursively forwarded

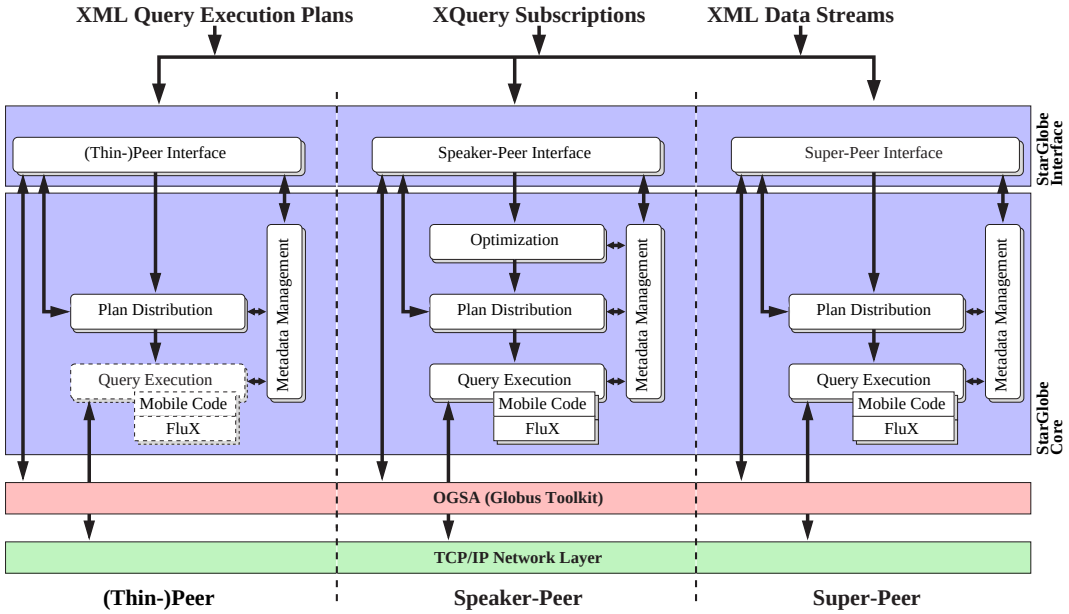


Figure 1.1: Overview of the StarGlobe interface. The *Mobile Code* (chapter 3) and *Plan Distribution* (chapter 4) components as well as the interface to *XML Execution Plans* (chapter 5) are contributed by this thesis (comp. [KSKR05]).

to all participating peers.

This process of integration in StarGlobe is done in two steps. First the plan is inserted manually into the network. Later on, the optimizer should perform this job.

To provide access to the different data sources, wrappers for the astrophysical catalogs have to be written.

The concept of mobile code (operators), which originates from the ObjectGlobe project, is implemented as well, yet with less focus on security and privacy issues, as e-science collaborations are assumed to take place in trusted environments.

Both query plan distribution and dynamic loading of user-defined operators will be implemented in the StarGlobe interface tier (see figure 1.1).

Chapter 2

Astrophysical Scenarios

In the context of e-science, we consider two scenarios where the benefits for the astrophysical communities motivate the usage of data stream management systems.

The first is *spectral energy distribution (SED)* classification. Classification of luminaries by means of spectral energy distributions is a central part in the work of the Max Planck Institute for Extraterrestrial Physics (MPE) in the GAVO project [GAV05]. Integration of data streams in the matching process promises to free potential for optimization.

The second example is a more visionary scenario about a network of robotic telescopes. Professionals collaborate with amateurs working on a global infrastructure connecting several telescopes around the world and satellites in the orbit.

To meet the needs of the communities which can be derived from scenarios as the following are the crucible by which an infrastructure for collaborative research will be judged.

2.1 Spectral Energy Distributions Classification

2.1.1 Classification of luminaries

The classification of *spectral energy distributions* (SEDs) is a key research area for the astrophysics community. It gives clues about the physical characteristics of celestial bodies, for example, whether they are *active galactic nuclei* (AGNs)[1], galaxies, or quasi-stellar objects (quasars).

What characteristics determine whether a star is classified "white dwarf", "bright giant", or "G"? The most prominent characteristic used certainly is the photospheric temperature. Table 2.1 shows the classification according to

[1] galaxies whose center emits irregular amounts of energy. The energy is freed by tidal forces of supermassive ("very heavy") black holes or stars which tear approaching stars apart.

Class	Temperature	Star Color
O	30 000 - 60 000 K	hottest blue
B	10 000 - 30 000 K	hot blue
A	7 500 - 10 000 K	blue, blue-white
F	6 000 - 7 500 K	white
G	5 000 - 6 000 K	yellow (like the sun)
K	3 500 - 5 000 K	orange-red
M	2 000 - 3 500 K	coolest red

Table 2.1: Morgan-Keenan spectral classification. A well-known mnemonic for these spectral types is "**O**h **B**e **A** **F**ine **G**irl, **K**iss **M**e" (adapted from [Wik05b, Ill79]).

Morgan-Keenan and as there is a succinct overview on the other classification models available at Wikipedia, we leave it by this short excursus [Wik05b].

Quality of classifications increase if a pan-chromatic collation of data or the combination different dimensions is used. The pan-chromatic approach combines photometric measurements from different frequency bands. As an example of combining measurements from orthogonal dimensions consider using photometric and gravitational data for classification.

Two kinds of spectra take an important role in the classification process:

- observational spectra
- theoretical spectra

The process to generate observational SEDs can be subdivided into SED-matching and SED-assembly. First potential useful information is collected from each catalog individually. The retrieved results are finally assembled (the information from the different catalogs is fused together) into the observational SEDs.

Theoretical spectra are the spectra especially designed to represent a celestial body with certain features (such as temperature or gravitation) and against which the observational spectra are matched. Before the classification can take place, these theoretical spectra have to be filtered, sampled, and modified to simulate an observation. Without these modifications the theoretical results are far to detailed or exact and the classification might fail due to this differences.

In general, users who want to conduct a classification have to provide two things: a list of *luminaries of interest (LOIs)* and a list of catalogs they want to retrieve the data from.

The list of LOIs, also called an *input list*, specifies the celestial coordinates and describes the positions the researcher is interested in. The coordinates are specified using the *right ascension (ra)* (comparable with the longitude of us earthlings) and *declination (dec)* (equivalent to latitude).

The second list specifies catalogs, archives, or similar data storages which will be considered to match the observed objects on their theoretical counter parts. It contains all necessary meta-data like the storage location, the access information, data types and schema of the content.

At the end of the classification, researchers get a ranking of the n most probable classifications for each LOI.

2.1.2 The A-Star Workflow

Adorf et al. describe the complete SED workflow and its implementation as an automated Web service as part of a virtual observatory [AKL+04]. In the following we go through a simplified scenario related to the SED classification which focuses on some of its core building blocks.

An astrophysical group is interested in *A-Star* (right ascension = $20^h\,48^m\,13.3°$, declination = $33°\,26'\,26''$). It performs a cone-search on the ROSAT All-Sky Survey Bright Source Catalog (RASS-BSC) to find all sources (entries in the catalog) within a radius of 0.5° around *A-Star* and is also interested in the distances to *A-Star* of the retrieved sources.[2] The RASS-BSC (or just RASS) is a catalog which is derived from a all-sky survey performed by the ROSAT mission between July 1990 and August 1991. With ROSAT ("Röntgenstrahlen Satellite"), the MPE conducted the first all-sky survey in X-rays [VAB+99].

> Retrieve all sources (stars) from the catalog RASS, that are within a radius of 0.5 degree from star *A-Star* and calculate the Mahalanobis distance, as well as the polar angle for those sources.

This kind of query is called a *Simple Cone Search*, followed by additional distance calculations.

To conduct a (simple) Cone Search on a catalog, one specifies a center (point on the sphere) and the search radius. Calculating this using Cartesian coordinates is much faster, thus a conversion step from spherical coordinates to Cartesian coordinates is necessary. To measure the distance between selected catalog sources and the center of the cone the Mahalanobis (a multi-dimensional variance scaled) distance is calculated.

Figure 2.1 shows a possible realization of these steps and a distribution of the operators across the network.

Our network contains four peers (*Peer* 0 through *Peer* 3) and one catalog originated at *Peer* 0. Whether the catalog is a database or just a comma-separated file accessible via the File-Transfer-Protocol remains transparent to users, and

[2] Among the astrophysical community *A-Star* is more commonly known as *1RXS J204813.3+332626*. That is far too long for a workflow title and we therefore use the short-form *A-Star*.

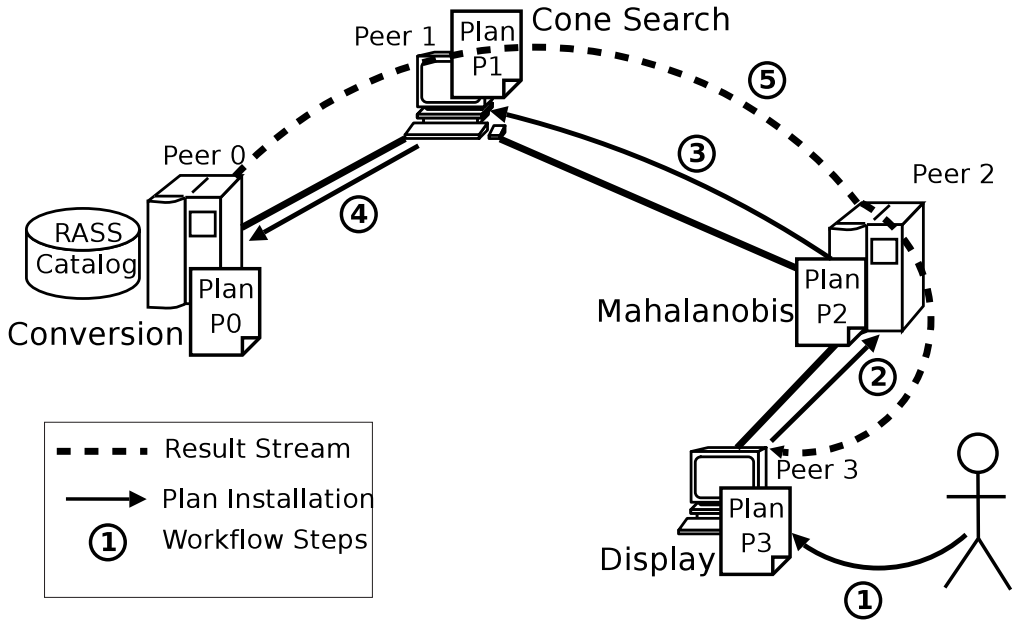


Figure 2.1: The A-Star workflow. A research queries all sources that are within a certain radius to A-Star. The user-defined operations are installed along the path towards the catalog.

so advanced access techniques as stored procedures or let alone user accounts to execute shell scripts on the data are not available.

The infrastructure does not provide implementations for the building blocks of the query but offers an interface for user-defined operations. The user implements three operators with the necessary functionality and provides them on a web server.

The user creates an XML-based query evaluation plan which specifies the operators and where they should be installed. Finally, the plan is injected into the network (step 1).

Each Peer receiving the plan installs the specified operators and propagates the sub-plans to its neighbors (step 2 - 4).

Then the catalog is processed and directed through the operator chain and the result is finally displayed at Peer 3 (step 5).

Even this simple example shows the potential for distributed parallelism and pipelining in the SED classification process and how it would benefit from using data streams.

2.2 Early Alerter Systems

Which little teenager has not gazed at the starry canopy in a starlit night to count as many shooting stars as possible?

Amateurs or professional star-gazers curiously study such volatile events like comets, asteroids, Gamma-ray Bursts (GRBs), or supernovae. To discover an event of unknown type and to open new fields for scientific research propels efforts in the astrophysics community to create a publish&subscribe architecture for such events.

Various notification systems already exist and are now tried to be standardized with the advent of new enabling technologies and the foundation of virtual observatories. The Central Bureau of Astronomical Telegrams (CBAT) [CBA05], the Gamma-ray Burst Coordinate Network (GCN) [GCN05], and the Astronomer's Telegram [ATE05] are just a few examples for such notification systems, partially still using natural language.

2.2.1 Automatic Follow-Up Observation

The IVOA Sky Event Reporting Metadata (VOEvent) specifies content and meaning of an XML based notification packet which aims at driving robotic-telescopes, triggering automatic archive searches, and to alert the community for various purposes [SWA+05]. The specification provides a lot of valuable insights about the current state and perspective of astrophysical notification services, even in its early stage of development.

Robotic telescopes will respond to these alerts with near real-time follow-up observations across different wavelength bands and extend the scientific coverage of such short-time events.

2.2.2 The Alerter Workflow

We envision a scenario where several globally distributed institutions federate in a common network. These sites provide archived data or contribute by publishing new data from their telescopes and satellites into the virtual network. Of course, sites can contribute to the network with both kinds of information. All sites use a data stream management system (like our StarGlobe infrastructure) to publish their data and to subscribe data from other catalogs. In addition to the result streams, there also exists a special stream on which events are transmitted. Every organization can configure which events it wants to listen to by installing a filter on that stream.

Figure 2.2 shows the visionary example of distributed collaboration that we now will describe in more detail.

At institution A new observational data is received (step 1). The data gets filtered against previously defined thresholds. If a new event is matched by the filter, it is a possible candidate for an unexpected event worth being monitored by several institutions. Yet before a false alarm is broadcasted to the other participating groups, results from previous observations are consulted (step 2).

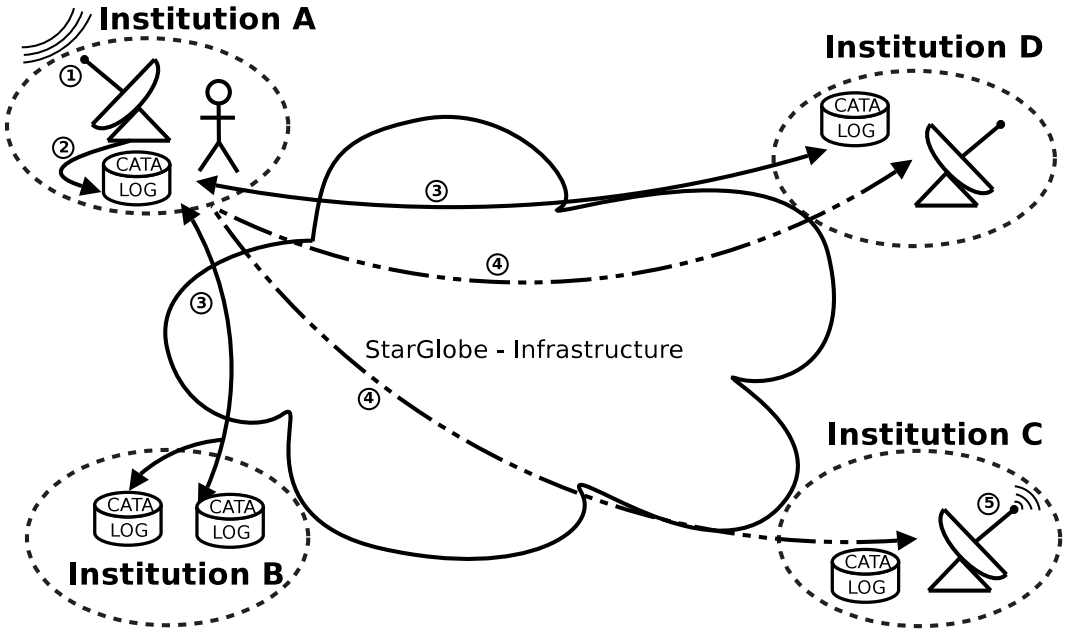


Figure 2.2: The Alerter Workflow. Distributed Organizations are interconnected by a notification system which runs on StarGlobe. If new data arrives (1) and comparisons with own (2) and distributed (3) data federations show that it is an interesting observation, an event is sent across the notification system (4) and automatic follow-up observations are performed (5).

If a comparable event has occurred before several times it is conceivably not that special.

If the query on the own catalog does not provide new information (step 3), the new measurement is compared with data gathered by other projects (step 3). As a consequence a search is placed on different data-sources in the federation. Provided that there is no similarity to measurements by other organizations, the observed event has a high probability of being extra-ordinary.

When this assumption holds, the region should be observed by as many observatories as possible, and institution A will send a notification to the other research facilities (step 4).

Of course, a decision whether to join into such observation is scheduled according to local priorities and whether the telescope is physically able (e. g., area not visible to telescope or temporal constraints) to support it. In our example only institution D joins into the observation (step 5).

By synchronizing many telescopes on a specific event observation time is increased from six hours in average (caused either by the night-day cycle or worsening observation conditions) to 24 hours.

2.3 Community Needs

These two examples show the needs that can be addressed by a data stream management system. As the relatively small number of catalogs is confronted with an increasing number of queries, stream sharing and in-network query processing will considerable reduce the network load.

Matching several catalogs against an input-list can be conducted in a distributed fashion and with high parallelism by balancing the load across several machines, even those machines which do not provide data sources.

The convergence of broadcasting events to collaborative organizations and the data stream paradigm is quite apparent and also the notion of continuous listening for events, that trigger follow-up observations fit well into the context of StarGlobe.

The individual steps in the catalog workflow as well as the processing of VOEvents are special-purpose operations that underline the need to open the StarGlobe interface to self-defined functionality.

Chapter 3

Mobile Operators

As described in section 2, astrophysical applications need special purpose functionality. Mobile operators will be the extension to the StarGlobe interface to support these operations on data streams. Having defined the interface for the operators, users can implement against it and integrate their functionality in query execution. Another fundamental aspect of external operators is openness for change. We may see and meet the needs of current users, but can we anticipate those of next-generation communities? Proficiency to support user-specific behavior can be crucial for the adaption of an infrastructure. Throughout the thesis the terms mobile, external, or user-defined operators or functions are used as synonyms.

3.1 External Stream Processors

When such a query is installed following a manually designed query plan, several operators are distributed among different peers. These operators may simply forward streams between interconnected peers or display results of calculations. But these calculations – and we focus mainly on that part – may be produced by user-defined operators.

So, what are the general properties of these operators or *stream processors*? Well, the obvious one is depicted in Figure 3.1: A stream processor transforms an input stream into an output stream.

Figure 3.1: Abstraction of a stream processor.

A stream processor can modify the structure of the incoming stream (e. g., adding additional data calculated inside of the stream processor) or it filters the stream using *content-based* (a selection) and/or *structure-based* criteria. A

projection filters the structure of a stream and modifies the resulting output stream by leaving out some parts of the data.

There exist already some stream processors in the StarGlobe infrastructure, for example forward, display, and query stream processors. Forward operators forward streams from one super-peer to one of its neighbors, display operators' functionality should be self-explanatory and query operators are generated from a WXQuery. *External stream processors* are a new kind of stream processor that support user-defined functionality.

3.1.1 Building Block Extraction

What is the content of a data stream? Several data items – *building blocks* as we call them in StarGlobe. However these building blocks can vary. Consider a data stream with the possible DTD fragment <!ELEMENT stream (a b | b c a | c)+ > that contains a b b c a c ... as building block sequence.[1]

Some properties of a building block make the difference whether we are interested into it or not. To identify building blocks is integral, especially for handling very long streams. With a very long stream it is almost certain that we missed the beginning of the stream. To operate on such a stream we need to inject a contrived stream tag before we start processing the stream. Allowing such streams as described in the previous paragraph is possible yet quite complex to find a valid building block. Imagine we start reading the upper building block sequence at the first c. We have to read so much data from the input stream until we find a valid insertion point for the opening stream tag. We have to match the building block sequence against all suffices for any building block. Unfortunately the third building block is contained in the second building block pattern and thus after reading the follow-up element a, we can be sure just having read a whole building block and can insert a building block. For the sake of simplicity we assume in the following our streams to be homogeneous.[2]

A stream processor can extract the building blocks of the stream and as soon as a building block is complete, it can be processed.

In StarGlobe, a building block is an XML fragment. A number of user-defined functions only need to extract some properties of the building block. Consider an exemplary selection on photons which have an energy pulse higher than 1.3 keV. The operator having this functionality either drops a building block (if it has too low energy) or writes it on the output stream of the stream processor.

3.1.2 XML Transparency

To calculate whether a energy pulse is above 1.3 keV knowledge about XML is not necessary, whereas for extract the information about the pulse from the

[1] The DTD fragment says that either the sequences a b, b c a, or c are valid building blocks.

[2] Homogeneous in the sense, that there is only one distinct building block for each stream.

stream it is. So how confident are programmers of user-defined functions with XML?

This cannot be said clearly. Researchers who want to use StarGlobe might be not confident using XML (maybe they are confident with using comma-separated vectors) and this motivates the reduction of XML a implementor of user-defined functionality is obliged to deal with. If users want more control over the data accessed by their component they can access building blocks directly. The latter approach though being more powerful is dependent on the representation between stream processor and mobile operator.

All external operators have common parts in their stream processing. They need to extract the building block, extract some parts of the building block, and eventually change some parts of it before it is written back on the output stream. The step where the "real" functionality is executed can be extracted into a separate component; a processing unit or a *stream iterator*.

After the basic ideas of the stream processor concept, the next sections cover what options for designing that stream iterator component exist, how it has been implemented and what conclusions can be drawn from the current implementation.

Generally, operators are not restricted to work only on one input stream. For complex queries sometimes the capability to join several streams is needed . Thus, a stream processor may have at least one input stream. Structural information about a stream is necessary to extract corresponding building blocks. This is provided by either a DTD or an XML Schema for each input stream and also for the output stream. The output-DTD is necessary as operators might be installed in sequence. How to do joins on streaming data is one of the major challenges for data stream management systems and deserves separate treatment.

To reemphasize, building a representation of the building block, extracting the data for the stream iterator and writing the building block (with additional data) back to stream are operations which are common to all external operators. So this functionality should be provided by a generic stream processor. Whenever we instantiate such a stream processor, we need to configure the data extraction mechanisms and probably send additional configuration parameters to the stream iterators. The interfaces between the processing unit and the embedding framework must be flexible enough, that it works with different processing units.

3.1.3 Processing the data stream

For processing the input stream we need a parser. As potentially infinite streams are used, model-parsers are inappropriate for the whole processing (e. g., the parsers using the DOM-model) as they would construct the whole document in-memory.

Thus a SAX-Parser (an event-driven or push-based parser) seems the right way to process the input stream. Bare SAX-Parsing is read-only. But for the

modifying operators we need a model which is capable of such modifications. That brings DOM-Parsers back into play as they are capable of constructing new elements.

So there are several options. The SAX-Parser constructs a DOM-Node for each data item and when returned from the processing unit it can be modified. It combines the effectiveness of the SAXParser and the structure-modifying capabilities of the DOM-model.

A different strategy can be the usage of so-called SAXFilters. Such a parser works like a ContentHandler (the class processing the parser events) to a parser and filters (or modifies) these events to a different ContentHandler. The idea is, to collect the events for one building block and extracting the data for the Stream Iterator. When the Stream Iterator has processed the data item, the events are passed (perhaps together with additional events for output of the Stream Iterator) to the last Content Handler which writes onto the output stream.

The design of the stream processor that provides the input for stream iterators should provide a framework that these different strategies can be transparent to the Stream Iterators. Defining an abstract class with an generic *event objects* realizes this. These event objects contain a representation of the current data item.

Three approaches offer different design trade-offs. Implementers either work directly on the event object or specify what properties of the building block should be extracted and be written back to the output stream. Certainly, the first approach keeps XML as transparent data structure and is suited for easier functions. Complex user-defined functions profit from the flexibility how a building block is modified, however are dependent on the implementation of event objects. The third option is to implement a stream processor providing the user-defined functionality without using the stream iterator mechanisms.

The concrete implementations described above could then use either a DOM-Node or a sequence of parser-events or implement the functionality without the Stream Iterator concept.

At this stage, the streams can be seen as a collection of building blocks and thus using a push-based variant of the *Iterator* design pattern [GRJV95] is adequate.

3.2 Implementation

The StarGlobe system runs on Java™ 2 Platform Standard Edition 5.0 [Sun05]. It uses several new features of the 5.0 release especially the java.util.concurrent package, however generics, and other new language features cannot be used as Globus Toolkit 3 uses the Axis[3] library from the Apache project which has an

[3] ws.apache.org/axis/

org.apache.axis.enum package. In the following we assume the reader is familiar with the syntax of the Java programming language and use it for source code examples.

The streamglobe.services.p2p.engine.operator package contains besides built-in stream processors and stream processors necessary for query processing also the parent type for all stream processors, the abstract StreamProcessor class and a factory (StreamProcessorFactory) to instantiate an individual stream processor. We integrated the handling of external operators in the starglobe sub package. The external stream iterators are implemented in the org.gavo.streamoperators package and thus are on the one hand "really" external and on the other hand acknowledge the fact that they rely on libraries provided by the GAVO team from the MPE.

3.2.1 The StreamIterator Interface

```
package streamglobe.services.p2p.engine.operator.starglobe;
import org.apache.commons.beanutils.DynaBean;
public interface StreamIterator {
    public void open(DynaBean config, StreamWriter writer);
    public void next(StreamIteratorEvent nextItem);
    public void close(String streamId);
}
```

Listing 3.1: The StreamIterator interface

The original iterator is a means to abstract from an underlying collection. After the initialization of an iterator, applications *pull* the next object from the collection. In the Java Collections Framework, a client of an iterator invokes its public Object next() method. Listing 3.1 describes the interface of stream iterators. A stream iterator gets the next building block as event object, which were discussed above and are realized in the interface StreamIteratorEvent. It provides a transportation means between stream iterator and the generic framework for embedding external functionality.

After processing a building block need a mechanism to send the event object back to the stream processor. The decoupling of passing the building block to and returning it from the StreamIterator is motivated by considering aggregate functions. In this context, several building blocks are processed before the aggregate gets passed back to the stream processor.

That led to the idea of introducing a StreamWriter interface and pass an associated instance to stream iterators during initialization.

3.2.2 The StreamWriter Interface

When a stream iterator has processed (or generated) data that should be written on the output stream, it sends a write(StreamIteratorEvent) message to its associated stream writer. Thus an aggregate function could store several items in an internal buffer, calculate the aggregation and only communicate with its writer. When the external operation needs to write new data to a building block, the StreamWriter writes data to the specified locations into the building block.

```
package streamglobe.services.p2p.engine.operator.starglobe;
public interface StreamWriter {
    public void write(StreamIteratorEvent item);
}
```

Listing 3.2: The StreamIterator interface

3.2.3 Parameter Passing and Extraction

```
package streamglobe.services.p2p.engine.operator.starglobe;

import org.apache.commons.beanutils.DynaBean;

public interface StreamIteratorEvent {
    public DynaBean getParameters();
    public void setParameters(DynaBean params);
    Object getEventData();
    void setEventData(Object o);
}
```

Listing 3.3: The StreamIteratorEvent interface

The parameters of individual stream iterators are unknown before such an operator gets installed into the system. The access paths to relevant data depends on the processed stream, and also the paths where to write the output variables of an stream iterator are not fixed. Therefore the set of configuration properties of a stream processor and the access paths have to be dynamically calculated.

APIs intended to simplify access to dynamically calculated sets of property values are provided by the BeanUtils[4] package and especially the *Dynamic Beans (DynaBeans)*. We use DynaBeans to configure StreamIterators and to provide transparent access to building blocks.

The Standard JavaBeans mechanism to access a property *myProperty* of type MyType generally happens with a getter MyType getMyProperty() and a setter void setMyProperty(MyType). The DynaBean mechanism uses a hash table to

[4] http://jakarta.apache.org/commons/beanutils/

access this property with getProperty("myProperty") and setProperty("myProperty", value). There are various DynaBean flavors and for our very dynamic context the *Lazy*DynaBeans suite best as offering late instantiation and dynamic property addition.

The org.apache.commons.beanutils package is used by the Apache Struts-Framework[5] and since 2000 developed in the Jakarta-Commons project. This indicates that support and development of the library will go on and stable functionality.

Another consideration is that the cooperating partners at the MPE already used DynaBeans in a different project and so the customers using the system are not obliged to learn a totally new mechanism.

We also looked at the XMLBeans[6] technology as it supports the generation of Java types from XML Schema and therefore seemed appropriate to generate the building block representations. Eventually, the fact that we are using DTDs and the existing user experience about DynaBeans ruled XMLBeans out for this purpose, although we see, that using DynaBeans introduce the necessity of down-casts, and thus possible exceptions during run-time.

In our prototype we have refrained from hiding the DynaBeans behind a self-defined interface (see Listings 3.1 and 3.3), which is useful to hide the dependency to this library. If DynaBeans are not appropriate in an other context, only the implementation needs to change.

The StreamIteratorEvent interface (see Listing 3.3) provides access to the building block representation itself for more complex stream iterators and implementers who are familiar with that specific implementation or the extracted parameters in a DynaBean.

3.2.4 Embedding Stream Iterators

A framework which provides the functionalities as input- and output-stream handling and building block extraction embeds the individual stream iterators. As building block representation we decided to implement the DOM-Node approach described in the previous section. This decision is reflected by the DomNode-StreamHandler and DomNodeEvent types. The former processes SAX events to a DOM-Node, wraps it into an instance of the latter class which implements the StreamIteratorEvent interface and is passed to the user-defined StreamIterator.

All vital information for the stream processor including the DTD information about the incoming and outgoing streams, configuration of the stream iterator and the stream iterator itself are combined into StreamProcessorConfig. These parameters are extracted from query execution plans. We use this kind of parameter object [Fow99] to keep classes interfacing with external stream operators

[5] http://struts.apache.org
[6] http://xmlbeans.apache.org

unaffected from changes in the concrete parameters. Additionally the StreamProcessorConfig separates plan representation and operator functionality. One could have used a kind of assembler as described in the *Data Transfer Object* design-pattern [Fow02] to increase the decoupling of these both domains, yet as the plan structure can be considered stable and therefore this has seemed like an unnecessary additional indirection for this implementation.

To map data between the streams and variables in the stream iterators we use final-state-machines in both directions. We describe this at an more comprehensive example when we look at the implementation of the astrophysical workflow.

In favor of simplicity of the prototype, we did not implement the feature, that users can implement a complete stream processor and integrate it into the system. We can address this issue by extending the StreamProcessorFactory to use reflection instead of conditionals, let the AbstractExternalStreamProcessor implement the StreamIterator interface as well. If a user then specifies an external operator in a query execution plan (which we cover in the next section) we can decide at run-time whether the type specified is a stream processor (and thus can be integrated without embedding framework) or just a stream iterator. In the latter case we would use a standard implementation like the one described above.

Figure 3.2 gives an overview about the classes described in this section.

3.3 Astrophysical Workflow

In this section we show the implementation details of three user-defined functions for the astrophysical workflow described as an motivating example in chapter 2. To achieve a reasonable succinctness of the source code shown, only important parts are shown in this chapter. The complete sources for the three covered operators are placed in appendix C.3. The course of description is equivalent to the order of execution in the specified workflow.

The first operator performs a transformation from polar coordinates into Cartesian Coordinates. This step is critical for the performance of the next operator: a simple cone-search. A cone-search (selecting points on a sphere which are within a certain radius to a given point) on astrophysical data is easier and more efficient when using Cartesian coordinates. To calibrate the cone-search we need a center and a radius. Eventually, the Mahalanobis distance is calculated. It is measured from all data points that passed the cone-search filter to the center of the cone-search. This gives a notion about how close the matching elements actually have been. In addition to that, the operator calculates the polar angle.

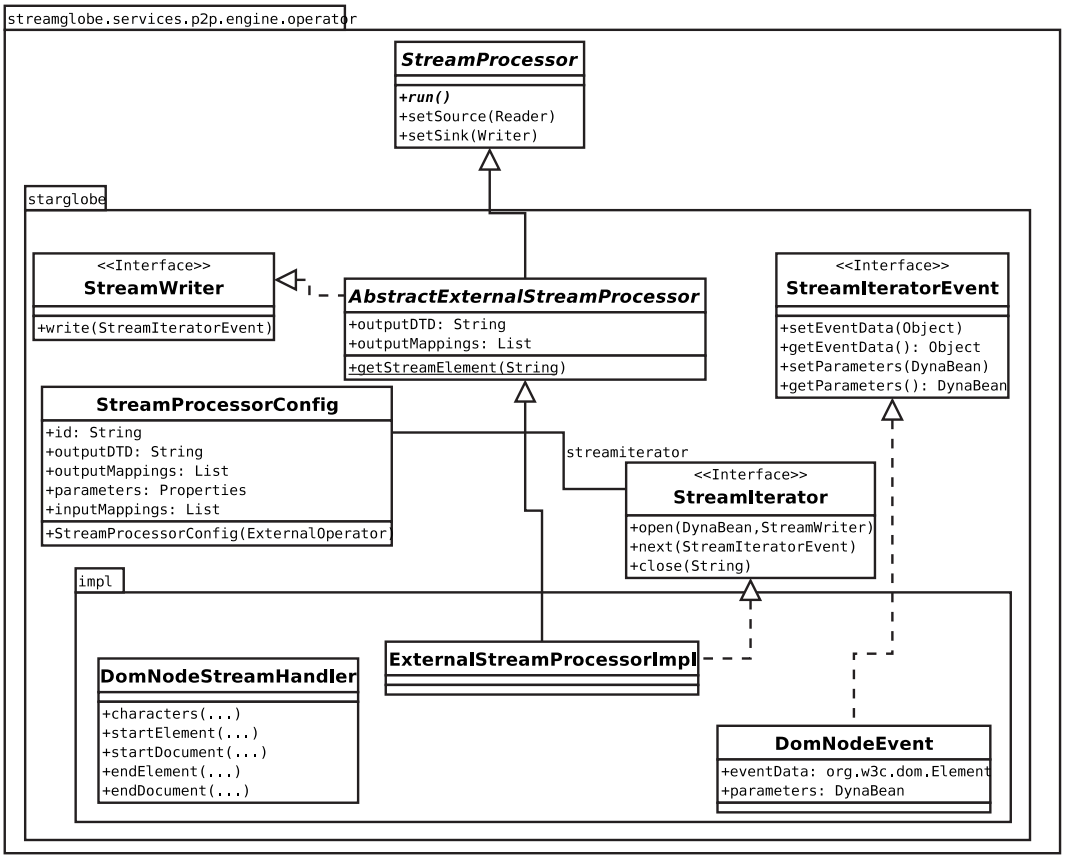


Figure 3.2: Package structure for external operators.

3.3.1 Coordinate Transformation

Given two polar coordinates α (longitude) and δ (latitude) on a sphere with radius R (resp. on the unit sphere), we get the Cartesian coordinates by the following equations.

$$r = R \cdot \cos\delta \qquad (\cos\delta) \tag{3.1}$$
$$x = r \cdot \cos\alpha \qquad (\cos\delta \cdot \cos\alpha) \tag{3.2}$$
$$y = r \cdot \sin\alpha \qquad (\cos\delta \cdot \sin\alpha) \tag{3.3}$$
$$z = R \cdot \sin\delta \qquad (\sin\delta) \tag{3.4}$$

The right ascension and declination are measured on the unit sphere [7] and therefore we can calculate the coordinates using the special case of the equations above.

As an example, we transform the coordinates from *A-Star* into Cartesian coordinates. So first we have to convert the right ascension $20^h\,48^m\,13.3°$ into degrees (its declination was $33°\,26'\,26''$). Right ascension is given in hours (24h) and having a range of 360° (15° per hour) we get approx. 312.05542 degrees right ascension and approx. 33.44056 degrees for declination. Inserted into the formulas above we get approximately $x_a = 0.55896$, $y_a = -0.61958$, $z_a = 0.55107$.

The implementation (after knowing the math) is straightforward and therefore an ideal example to start with looking at functionality of a stream iterator. As the main functionality lies in the next(StreamIteratorEvent) method we only show a fragment of the implementation.

```
public class CoordinateStreamIterator implements StreamIterator {
    ...
    public synchronized void next(StreamIteratorEvent nextItem) {
        DynaBean params = nextItem.getParameters();
        Double ra = (Double) params.get(RA);
        Double dec = (Double) params.get(DEC);

        double[] cartesian = CoordinateTransformation.toCartesian(
                    ra.doubleValue(), dec.doubleValue());

        for (int i = 0; i < cartesian.length; i++) {
            params.set(CARTESIAN, i, String.valueOf(cartesian[i]));
        }
        writer.write(nextItem);
    }
}
```

[7] using a certain reference system, currently *J2000*.

Listing 3.4: User-defined coordinate transformation.

The extraction of the right ascension and declination values and the insertion of the Cartesian coordinates into the stream is executed by the ExternalStream-ProcessorImpl class. The mappings are specified in the query execution plan and listing 3.5 shows the relevant part of the coordinate transformer configuration (omitting the DTDs).

The *variable*-Tags have two functions. For the input stream they specify which data should be loaded into the variable of the given type. The *mapping paths* (the select-attribute) are relative to the building block. The external stream processor builds for the building block and each input variable an automaton. It starts the building block-automaton when the input stream begins and as soon as the startElement event for the building block has been processed the product-automaton for the variable-paths are started and each time a processed element matches a step in the path the according automaton changes its state. When the opening tag for the last step in a mapping path is processed (e. g., for the variable RA it is the opening tag of the ra element. In this state a buffer is initialized and all following characters-events are stored in a buffer until the according endElement event has arrived at the content handler.

The implementor of the stream iterator defines mappings (i. e., keys), which are specified in the plan to retrieve either configuration parameters or data from the building block. As data types to initialize variables of stream iterators currently all wrapper classes for the primitive types (i. e., Integer, Double, Boolean, ...) are supported. However, this is extensible to any class which provides a constructor with a String as parameter. This classes then can be instantiated using the Java reflection mechanism.

```
<streamoperator id="transform"
  name="org.gavo.streamoperators.CoordinateStreamIterator"
    <variable name="RA" select="./coord/cel/ra" type="Double"/>
    <variable name="DEC" select="./coord/cel/dec" type="Double"
        />
  </inputstreamdata>
  <outputstreamdata>
    <variable name="cartesianCoordinates[0]" select="./
        cartesian/x"/>
    <variable name="cartesianCoordinates[1]" select="./
        cartesian/y"/>
    <variable name="cartesianCoordinates[2]" select="./
        cartesian/z"/>
  </outputstreamdata>
</streamoperator>
```

Listing 3.5: Coordinate transformer configuration from query execution plan (fragment).

3.3.2 Simple Cone-Search

Conducting a simple cone-search using transcendental functions is computational intensive. Basically, a *cone-search* query on a sphere looks for points p around a center c within a radius r. This can be imagined as follows. One takes the tangential plane in c and pushes it towards the center of the sphere. The intersection of this plane with the sphere resembles a circle with increasing radius. We push, until the distance between c and the border of the circle is exactly r. The clue is, that all points above the "pushed plane" fulfil the query, the others do not. It turns out that having the Cartesian coordinates reduces this to whether the dot product of the $c = (x_c, y_c, z_c)$ and $p = (a, b, c)$ is greater than the cosine of r. Equation 3.5 describes the formula of the simple cone-search.

$$a \cdot x_c + b \cdot y_c + c \cdot z_c > \cos r \tag{3.5}$$

This operator is another example for trigonometric mathematics, typical for astrophysical applications.[8] The next-method implements equation 3.5. The filtering function is performed by only sending the StreamIteratorEvent back to the ExternalStreamProcesser if the processed item qualifies. The center and search radius are initialized in the open-method.

```
public class SimpleConeSearchStreamIterator implements StreamIterator {
    ...
    public void open(DynaBean config, StreamWriter writer) {
        center[0] = Double.parseDouble((String) config.get(CENTER_X));
        center[1] = Double.parseDouble((String) config.get(CENTER_Y));
        center[2] = Double.parseDouble((String) config.get(CENTER_Z));
        Double radius = Double.valueOf((String) config.get(SEARCH_RADIUS
            ));

        maxDistance = Math.cos(radius.doubleValue());
        this.writer = writer;
    }

    public void next(StreamIteratorEvent nextItem) {
        ...
        double angle = SphericalTrigonometry.arcDistance(center, point);

        if (Trigonometry.cos(angle) > maxDistance) {
            writer.write(nextItem);
        }
```

[8] Jim Gray et al. address in a technical report how cone-searches and other spatial data searches can be realized in relational algebra and how database techniques can propel responses to these queries significantly [GSF+04].

```
    }
}
```

Listing 3.6: User-defined simple cone-search.

3.3.3 Mahalanobis Distance

The SimpleMahalanobisDistanceIterator comprises the final steps of the small workflow. It computes the Mahalanobis distance and the polar angle between the center of the cone-search and the data-item in the stream. This gives the user a measurement how good the actual matches from the data stream are.

```
public class SimpleMahalanobisDistanceIterator implements StreamIterator
    {
    ...
    private double[] refPointCartesian;
    private double[] refPointSpheric;
    private double sigma = 1.0;
    private StreamWriter writer;
    public void open(DynaBean config, StreamWriter writer) {
        List referencePoint = new ArrayList();
        referencePoint.add(Double.valueOf((String) config.get(REF_X)));
        referencePoint.add(Double.valueOf((String) config.get(REF_Y)));
        referencePoint.add(Double.valueOf((String) config.get(REF_Z)));

        refPointCartesian = getDoubleArray(referencePoint);
        refPointSpheric = CoordinateTransformation.toSpherical(
            refPointCartesian);

        if (config.get(SIGMA) != null) {
            sigma = Double.parseDouble((String) config.get(SIGMA));
        }
        this.writer = writer;
    }
    ...
    public void next(StreamIteratorEvent nextItem) {
        DynaBean parameters = nextItem.getParameters();

        double[] cartesian;
        double[] spherical;
        List pointCartesian = (List) parameters.get(POINT);
        cartesian = getDoubleArray(pointCartesian);
```

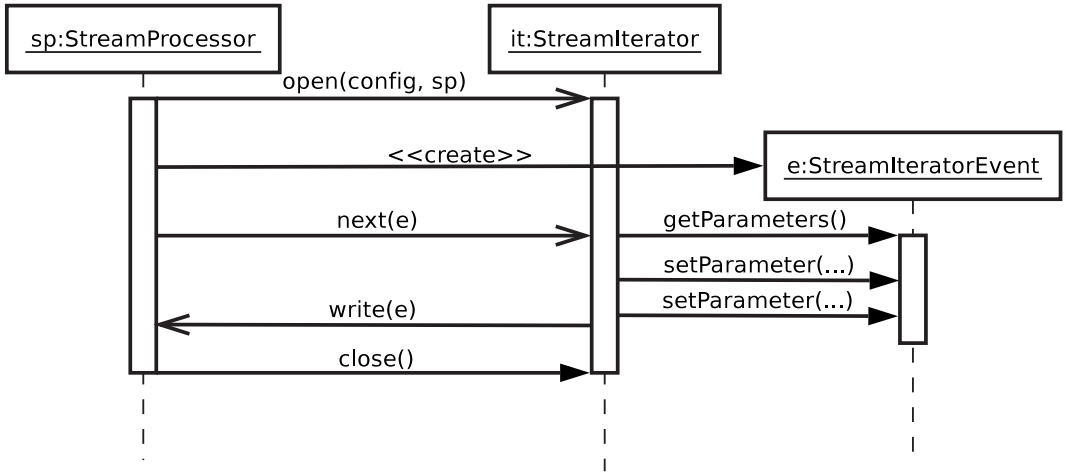


Figure 3.3: Communication between stream processor and stream iterator.

```
        List sphericalCoords = (List) parameters.get(SPHERIC);
        spherical = getDoubleArray(sphericalCoords);

        double arcDistance = SphericalTrigonometry.arcDistance(
            refPointCartesian, cartesian);
        double mahalanobis = arcDistance / sigma;
        double polarAngle = SphericalTrigonometry.positionAngle(
            refPointSpheric, spherical);

        parameters.set(DISTANCE, String.valueOf(mahalanobis));
        parameters.set(POS_ANGLE, String.valueOf(polarAngle));

        writer.write(nextItem);
    }
}
```

Listing 3.7: User-defined mahalanobis distance.

In general case the processing of data items is implemented in the next(StreamIteratorEvent) method. It reads the input parameters from the provided parameter object, calculates the result and writes the results back into the predefined parameters. Eventually, the processed data item – if it belongs to the result set – is passed to the StreamWriter to write it on the output stream.

The sequence diagram 3.3 sums up the interaction between stream iterator, stream writer, and external stream processor.

3.4 Integration of External Operators in FluX

To integrate user-defined functions into FluX a special namespace udf is reserved. Let's assume our external operator is implemented in the class streamglobe.udf.Average, this external operator could be integrated into the FluX engine using udf:streamglobe.udf.Average($p/price).

As in the stand-alone case, external operators need to handle sequences of elements, coming from the input streams. When the extraction of the data relevant was done by parameters in the plan specification this job is done by specifying the selection paths in the function call of the XQuery. The buffers or sequences are sent by the FluX engine to the user-defined function.

As the handling of DTDs is done by the FluX query engine and the external operators only need to handle the buffers or sequences which are returned from the engine.

When the FluX Engine is capable of returning multi-valued results (e. g., Coordinate conversion ra, dec results in x, y, z) external operators could be integrated in the query engine.

The class loader will load the operator class file before the query engine is started. So the engine can just instantiate the class as if it was not dynamically loaded.

Chapter 4

Query Execution Plans

With ObjectGlobe [BKK+01], a system for distributed query processing for persistent data on the Internet was introduced. ObjectGlobe integrates machines which provide processing power (*cycle providers*), distributed data sources (*data providers*), and repositories offering functionality which can be dynamically loaded during query processing (*function providers*).

In distributed systems with data publication and data subscription a schema is necessary to organize data. In the context of distributed databases a global – not necessarily central – schema is designed, besides an allocation schema (where and how data is distributed), and also a local schema which describes how data is organized on a single installation [KE01].

In the StarGlobe data stream management system users can register streams (induce the data into the architecture) and query those registered streams. The StarGlobe infrastructure offers two interfaces to specify transformations on data streams. The registerQuery interface enables optimization through the optimizer and thus reuse of preexisting streams. If users have specific knowledge about the application context, our system should enable them to use it by specifying query execution plans on their own. This is possible with the registerPlan interface.

When a SQL query is send to a RDBMS the query optimizer composes the best plan according to a specific cost model. For example, it decides which join implementation is the best, or whether the table should be accessed via an index or a full-table-scan.

When a subscription is injected into the StarGlobe system it must be determined where to install the operations (or operators) into the network to fulfill the query in an efficient manner.

4.1 Query Execution Plan Requirements

The StarGlobe system provides two separated interfaces to subscribe data streams: queries and query execution plans.

Queries are registered by sending the WXQuery from thin-peer to the connected super-peer. The speaker-peer optimizes these according a specific cost model and then creates a plan which is installed at the super-peer of the user.

Under certain circumstances users might prefer to layout the execution plan on their own. For example, our support for user-defined functions is at first depending on a manual injection.

Especially in the context of stream-widening with regards to multi-query optimization, operators running at one station can be moved from one machine to another (if it is too selective to support a newly injected subscription) or be removed totally.

On the other hand, the optimizer also should generate query execution plans and install them into the network. Thus it becomes evident that using the same format for the user-installed query execution plans as well as for plans generated by the optimizer would decrease system complexity.

After this considerations we present some of the fundamental requirements for the query execution plans.

One Plan for Each Query. Looking at the ordinary system, each calculation of a plan is triggered by the injection of a query. Therefore each plan should represent a single query. However one can integrate sub queries into a single plan which implies to use a recursive structure for the representation.

Lightweight XML. At the early stage of the system, users must write their plans manually until they are supported by a graphical editor or similar means. Consequently a plain text format seems appropriate. Having a validation of the plan can be helpful in two ways. It can support the user during the editing process and it can track down syntax and maybe some semantical errors before installing a plan in the system. These were the arguments for using XML as description language and to validate the expected structure using XML Schema.

Displaying Dependencies. A plan should make dependencies clear. A plan can depend on the availability of a certain operator, rely on a query already running in the system, or a sub query defined in itself. Some of these dependencies can be checked with the referential integrity mechanisms of XML Schema others can only be validated during runtime.

Easy Subplan Extraction. When a plan is installed at a peer it contains several information classes. We distinguish between local information (the operators which need to be installed at the local peer) and external information (subplans which must be installed on neighboring peers). To reduce message size and make it transparent to the peers whether the plan was injected directly at them by a user or by the optimizer at the speaker-peer, the extraction of the

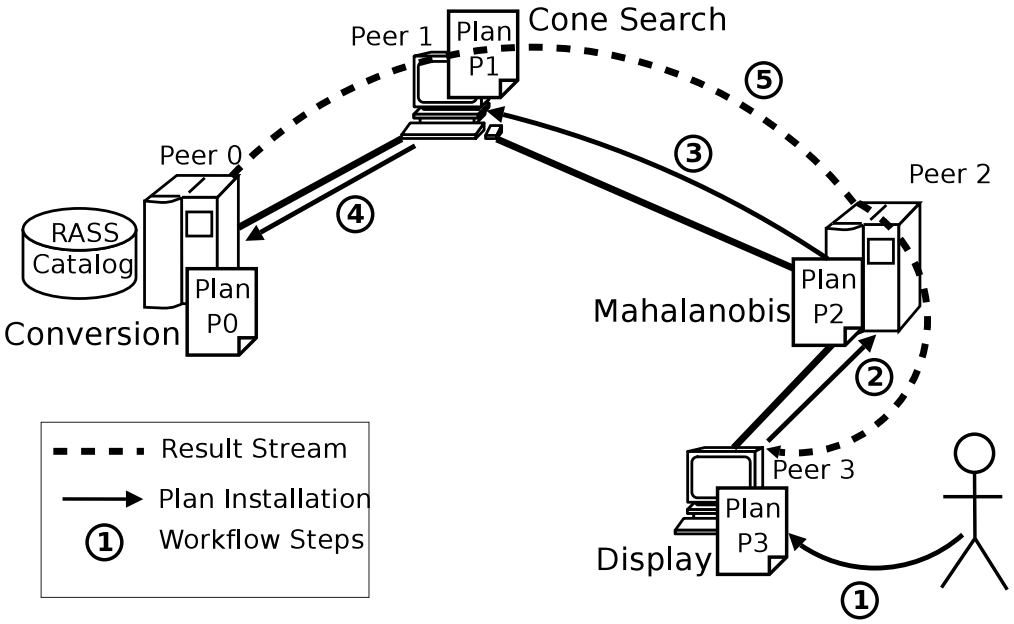


Figure 4.1: The A-Star workflow.

relevant subparts should be straightforward and each plan should be more or less self-contained.

4.2 Examples of Plans

Having shown the facets of our plan structure we look at two comprehensible examples of the query plan. They present the proficiency of StarGlobe supporting user-defined operations as well as how the plan serves the optimizer as means to disseminate the operators across the network.

4.2.1 Executing an XQuery

The first plan is quite a simple one. Figure 4.1 again shows the first astrophysical scenario, and now lets assume, that the catalog at $Peer - 0$ is at the MPE and the catalog contains data from the ROSAT satellite mission, especially the ROSAT All-Sky Survey (RASS). An astronomer from the MPE is interested in the Vela Supernova Remnant, which can be found roughly in the area of the right ascension being between 120 and 138 degrees and the declination between -49 and -40 degrees. Why an astronomer could be interested in that particular area, we cover in more detail in chapter 6.

So the query is installed at the same peer as the catalog resides and therefore the plan for answering this query looks like listing 4.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<plan atPeer="@Peer0-GSH@" id="query-1" xmlns="urn:streamglobe.in.
    tum.de/pdc"
  xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
    <streamoperator id="query-0" name="query"
      xsi:type="queryStreamoperatorType">
      <dependencies>
        <stream id="stream-0"/>
      </dependencies>
      <source><![CDATA[
          for $p in stream("stream-0")/photon
          where $p/coord/cel/ra >= 120.0
          and $p/coord/cel/ra <= 138.0
          and $p/coord/cel/dec >= -49.0
          and $p/coord/cel/dec <= -40.0
          return $p
          ]]></source>
    </streamoperator>
  </add>
</plan>
```

Listing 4.1: The query execution plan for the Vela-XQuery.

The query is dependent on stream-0 which is already available in the StarGlobe system and the WXQuery declaration is embedded in the source element of the query operator.

4.2.2 Executing the A-Star Workflow

Listing 4.2 shows the overall structure of the manually edited plan for the A-Star workflow. The plan structure starts at *Peer 3* and includes all plans for the other peers.

Only relevant subparts of the plan are shown in this chapter, yet the whole plan can be found in appendix C.

```
<?xml version="1.0" encoding="UTF-8"?>
<plan atPeer="@Peer3-GSH@" id="stream-4" xmlns="urn:streamglobe.in.
    tum.de/pdc"
  xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
    ...
  </add>
   <plan atPeer="@Peer2-GSH@" id="stream-3"
     xmlns="urn:streamglobe.in.tum.de/pdc"
     xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <add>
```

```
      ...
    </add>
    <plan atPeer="@Peer1-GSH@" id="stream-2"
    xmlns="urn:streamglobe.in.tum.de/pdc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <add>
        ...
      </add>
      <plan id="stream-1" atPeer="@Peer0-GSH@"
      xmlns="urn:streamglobe.in.tum.de/pdc"
        xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <add>
          ...
        </add>
      </plan>
    </plan>
  </plan>
</plan>
```

Listing 4.2: The A-Star query execution plan.

The complete workflow is executed on four peers and gets installed at *Peer 3*. Each plan contains the directives to add operators to the local query engine and an optional subplan. This optional subplan is extracted and send to the appropriate neighbor. In this specific example *Peer 3* extracts the subplan of *Peer 2* and sends the plan to this peer. Eventually, *Peer 0* is reached and as its subplan does not contain further subplans, it can instantiate the operators specified in the add part of its plan.

```
        ...
        <streamoperator id="transform" codebase="@GAVO_JAR@"
          name="org.gavo.streamoperators.CoordinateStreamIterator"
          xsi:type="externalStreamoperatorType">
          <dependencies>
            <stream id="stream-0"/>
          </dependencies>
          <authorizedby>Tobias Scholl</authorizedby>
          <inputstreamdata id="stream-0">
            <dtd>
            <![CDATA[
            <!ELEMENT photons (photon)*>
            <!ELEMENT photon (coord, phc, en, det_time)>
            <!ELEMENT coord (cel, det)>
            <!ELEMENT cel (ra, dec)>
            <!ELEMENT ra (#PCDATA)>
            <!ELEMENT dec (#PCDATA)>
            <!ELEMENT det (dx, dy)>
            <!ELEMENT dx (#PCDATA)>
            <!ELEMENT dy (#PCDATA)>
            <!ELEMENT phc (#PCDATA)>
```

```
    <!ELEMENT en (#PCDATA)>
    <!ELEMENT det_time (#PCDATA)>
    ]]>
    </dtd>
    <variable name="RA" select="./coord/cel/ra" type="Double
        "/>
    <variable name="DEC" select="./coord/cel/dec" type="
        Double"/>
  </inputstreamdata>
  <outputstreamdata>
    <dtd/>
    <variable name="cartesianCoordinates[0]" select="./
        cartesian/x"/>
    <variable name="cartesianCoordinates[1]" select="./
        cartesian/y"/>
    <variable name="cartesianCoordinates[2]" select="./
        cartesian/z"/>
  </outputstreamdata>
</streamoperator>
...
```

Listing 4.3: The coordinate transformer of the A-Star query execution plan.

Listing 4.3 shows the stream operator part of the plan for *Peer 0*. It specifies the necessary meta-data to install and run the stream iterator. The codebase attribute specifies the archive for the operator. The inputstreamdata contains the variables to retrieve the right ascension and declination as Double values. The automata to retrieve parameters from the stream are generated from the DTD in the inputstreamdata. The outputstreamdata contains the paths where the Cartesian coordinates will be stored to.

The specifications of the external operators for cone search and Mahalanobis distance are similar to the coordinate transformation. Listing 4.4 only shows a short fragment of the configuration of the cone search operator. The Cartesian coordinates of A-Star (as calculated in 3.3.1) and the search radius from section 2.1.2 are specified in key value pairs.

```
<parameter>
  <key>CENTER_X</key>
  <value>0.5589609245067093</value>
</parameter>
<parameter>
  <key>CENTER_Y</key>
  <value>-0.619582747878259</value>
</parameter>
<parameter>
  <key>CENTER_Z</key>
  <value>0.5510715955356713</value>
</parameter>
<parameter>
  <key>SEARCH_RADIUS</key>
```

```
        <value>0.5</value>
      </parameter>
```

Listing 4.4: The cone search of the A-Star query execution plan (fragment).

The last operator is the display operator at *Peer 3*. The configuration of this built-in operator just contains the dependency on the plan executed at *Peer 2*.

Having described the plan structure at two examples, a more abstract definition follows.

4.3 Plan Structure

The valid structure of a query execution plan is described with XML Schema. The only top-level element and thus the only valid root element of XML documents validated against this XML Schema document is the plan Element. The structure of this element and the other sub elements are described either as types or using in-line definition.

The full XML Schema of the query execution plans can be found in appendix B.

4.3.1 Top-Level Plans

The plan element describes all the operations at one peer which are necessary to process the query. Operators might be added or deleted at this peer and the installed operators can depend on other subplans installed at other peers.

As attributes the plan element has atPeer, an xsd:string, which is the Grid Service Handle of the Peer. The Grid Service Handle is the URL where the Grid Service can be addressed. The id (type: String) attribute is an identifier of the plan which should have either a stream identifier (stream-#) or a query identifier (query-#).

Structure

A query execution plan should at least add operators to a peer or delete them from a peer. Unfortunately XML Schema lacks the possibility to describe a pattern like (a — b — ab) without violating the *Unique Particle Attribution* constraint [TMM05]. As a consequence we needed to make the add and delete Element both optional and check for the occurrence of at least one of them in the implementation.

The structure of a plan element is a sequence of add (optional), delete (optional), and zero or more plan elements. The current prototype implementation does not implement operator removal, so currently the delete element is not processed.

Identity constraints

XML Schema extends the preliminary support of identity-constraints of DTDs using ID and IDREF(S) attributes to a more sophisticated mechanism using xsd:key and xsd:keyref elements.

In a plan there are several identity constraints, and by defining them in XML Schema we can assume their validity in our application code.

The first identity constraint is the uniqueness of operator descriptors and plan identifiers. Installed operators can refer to other operators or subplans executed on neighboring peers. The most flexible way to reference to operators on other machines would be an operator-level reference mechanism. To achieve this one would need to make all operators installed at a single peer available using result dispatchers. To find a trade-off between flexibility and reducing the number of threads, we decided only to make the output of the top-most operator installed at a peer reachable via the identifier of the plan.

To guarantee the successful resolution of references to either same-peer operators or subplans the identity constraint planIdOrOperatorId defines the valid range for these attributes. The referential integrity is controlled with the validReference constraint.

In the case operators are also deleted, the unique-constraint doNotDeleteNewOperators defines uniqueness on operator names in the whole plan. This avoids deletion of operators which were just installed by this plan. Perhaps, this constraint is too restrictive, when only operators are added. Distinct operators with same identifiers on different machines do not interfere.

4.3.2 Adding Operators

The structure of the add element is a sequence of one or more stream operators. These operators will be installed at the peer whose Grid Service Handle is specified in the surrounding plan element. There are several kinds of stream operators and they will be described in section 4.3.5. The XML description of the stream operators must be a subtype of abstractStreamoperatorType.

4.3.3 Deleting Operators

Again, the content of deleteOperatorsAtPeerType, which describes the structure of delete elements is related to addOperatorsAtPeerType. It is composed from a sequence of one or more stream operators. The reason to distinguish between two types is that to set up an operator more meta data is necessary than to delete it. In the latter case only the id of the operator is sufficient to find it and stop its execution.

To show the connection to the add element we use “streamoperator” as name for the sequence elements but use different structures mirroring the different

amounts of information needed for the complementing tasks.

4.3.4 Stream References

The input stream of stream operators can be twofold, namely a checked or unchecked source. On the one hand it can be the output stream of an operator at the same peer or a stream which is generated by a plan executed on the neighboring peer. These are the input sources that are validated with the integrity constraint defined in the plan Element (see 4.3.1). These references are specified with the streamreferenceType.

Streams or subscriptions can be also referenced using an element of the streamType, yet without the luxury of being checked valid using XML Schema techniques. The reachability of the identifiers specified have to be verified by the user (when injected manually) or the optimizer (when generating a query execution plan). An error in this type of source can only be detected during run-time.

4.3.5 Stream Operators

The abstractStreamoperatorType can be used for two purposes. On the one hand in the add element only this generalization is referenced and how the individual implementations are structured is not of importance in that context. On the other hand, information about stream operators which is needed for every implementation is extracted into this type.

Looking at stream operators from a high level of abstraction they need to define an identifier (so that they can be referenced or deleted later on) and specify all their dependencies.

Currently, the StarGlobe system only supports single dependencies (i. e., only one input stream) resulting in a linear execution plan. The extension to support multiple input streams from the "plan perspective" is obviously to set the maxOccurs attribute to unbounded. Implementing the real capability to handle multiple input streams deserves – as being far from straightforward – is a master's thesis on its own right.

Built-In Operators

The basic built-in operators of the system have a very special feature that distinguishes them from other operators. First, they do not need *any* configuration besides that for any operator: the dependency. The forward, display, and (meanwhile) obsolete null operators just forward the stream which is plugged into them. They especially do not have notion about building blocks of a stream. The forward operator makes streams at a peer accessible for neighbors. The display operator is the last operator in an execution plan and displays the result at the

last super-peer which injected the subscription. How the display operator can be replaced by the introduction of thin-peers is described in chapter 5.

Query Operators

The queryStreamoperatorType is the representation of a WXQuery injected into the system. The source element contains the WXQuery, the optional elements input-dtd and output-dtd specify the structure of the input and the output stream respectively.

Whether the input-dtd is integrated into the plan is subject to trade-off considerations. If it is integrated, super-peers do not need to contact the speaker-peer in most cases to resolve the DTD. When integrated into the plan the overall number of messages during the query installation process decreases, yet the message size increases (not considering the overhead of wrapping the messages into SOAP envelopes).

Another option could be to enrich the plans in a recursive fashion as the subplans get installed and subplans return their DTD to the next plan in the hierarchy.

In general it would be possible to enhance a WXQuery with user-defined functions as well. Parameters which are required are the code base (a repository in the sense of a function provider) where an archive containing the compiled Java classes can be retrieved from and the name of a class which implements the user-defined function.

The ability of using user-defined functions in queries themselves depends on the query engine and thus was not in the focus of this work. However, the design of the interface and the integration of mobile operators is comparable to the integration of external operators and is discussed in section 3.4.

External Stream Operators

The meta-data about external stream operators is defined in externalStreamoperatorType. From the variety of *operator repositories*, which provide the system with user-defined special purpose implementations, we support local or web-based repositories. We have chosen these two options for several reasons. Web-based repositories enable users to share their implementations of operators throughout the infrastructure in a convenient way. The compiled stream iterator can be made available on a web server, e. g., by compressing all necessary classes into a Jar archive. Additionally we support repositories situated at the local file system. The name attribute specifies the Java Type whose Class file implements the StreamIterator interface and can be loaded from its codebase. The restriction on the http- and file-protocol is defined in codebaseType. The attribute authorizedby is currently not used, yet can specify the distinguished name of the certificate used to sign the jar-File.

As stated beforehand, we generally assume trusted and collaborative environment for the operators. Should the importance of security issues rise, we could require the jar-files to be signed with the credentials provided by the security services of the Globus-toolkit. When the authorizedby attribute adheres to the DistinguishedName patterns described in RFC2253[1] it could be used by the CA of the Globus instance to verify the signature.

Input stream and output stream specify a dtd element and also provide a mechanism to select parts of the individual building blocks of the input stream inputstreamdata and to write it back to the output stream outputstreamdata. An external operator retrieves data from the input stream and can write data back on the output stream. To map between stream content and local variables of the user-defined function typedVariableMappings and untypedVariableMappings are used.

The latter specify the path where new data calculated by the mobile operator should be written back to the stream. Therefore the user does not need to specify type information as the data is serialized into XML. When deserializing data from XML into variables for the external functions type information is very useful and therefore the user can specify these.

4.3.6 Extending the Operator Hierarchy

In retrospective, the name "built-in operators" has been misleading as its intention is to subsume all operators that only need the stream identification for their input stream. Built-in, meaning "provided by the infrastructure", could be another unintended interpretation.

When a new operator type is developed, having its own kind of configuration, it is perfectly safe to create a new complexType which extends from the abstractStreamoperatorType.

We also decided not to include any of the general extension features provided by XML Schema as xs:anyAttribute or xs:anyElement. From our perception, gaps between flexibility in the plan and realized implementation would lead to a blurred conception of the real functionality by developers and people using the system.

4.4 Plan Implementation

The classes related to the representation of external operators in the overall plan structure is shown in figure 4.2.

The ExternalOperator is a subclass of the generic OperatorElement which has an operator identifier and dependencies. This class is basically the equivalent to the abstractStreamoperatorType. untypedVariableMapping and typedVariableMapping are realized with ParameterMapping and TypedParameterMapping respectively.

[1] http://rfc.net/rfc2253.html

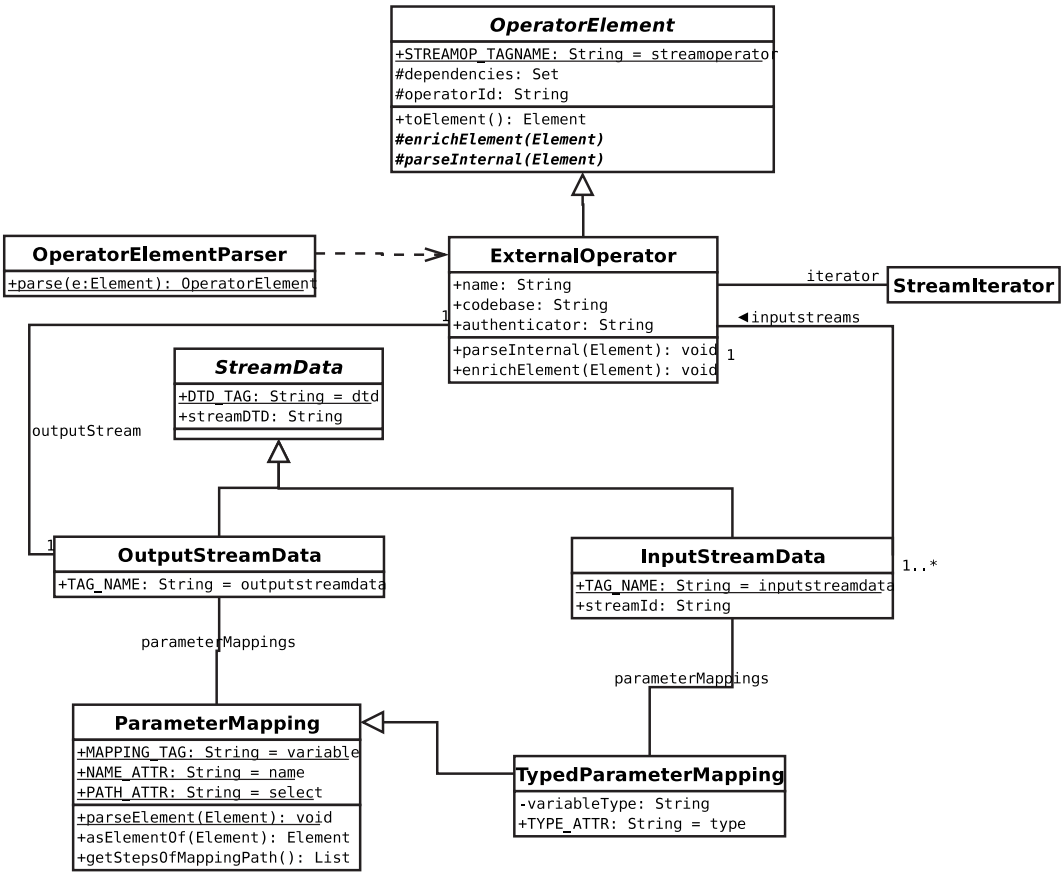


Figure 4.2: Classes of external operators in plans.

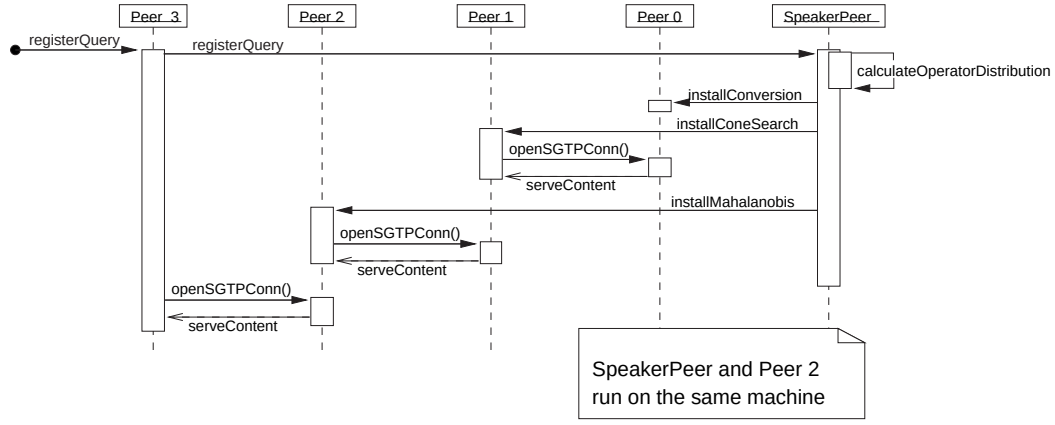


Figure 4.3: Centralized Plan Execution forces the speaker-peer into the center of communication.

4.4.1 Plan Distribution

For illustration we still consider the network of the A-Star workflow with the four super-peers. At super-peer $Peer0$ we inject a query on stream $s1$, which is installed at super-peer $Peer3$. $Peer2$ also takes over the part as speaker-peer. How do we disseminate the workflow in the system?

We compare a centralized approach without a plan with a decentralized plan-based technique.

The speaker-peer calculates what stream processors are necessary and invokes their installation. As soon the operator is installed at a super-peer, it establishes a SGTP connection to retrieve the necessary data stream.

When looking at the sequence diagram (figure 4.3) of that communication process the speaker-peer is obviously in the center of communication. This has the advantage that all the processing is at one place. Unfortunately this leaves all the work at the speaker-peer which is also responsible for calculating the best plan and also reduces the possibility to change or extend the installation process at peers.[2]

The sequence diagram (figure 4.4) shows how individual peers interact when the query plan is installed at the peer of the subscriber and this installs the subplans at the according neighbors. When installing a query execution plan on a peer two tasks have to be accomplished. The operators for this specific peer (specified in the `add` element) as well as the descending subplan must be sent to the according neighbor.

As the sequence diagrams show the interaction between the individual components in the approach is more centralized than in the second. Martin Fowler discusses the advantages and disadvantages of centralized control on a similar example in UML Distilled [Fow03], which can also be applied on this example.

[2]Possible extensions include quality of service or accounting and billing.

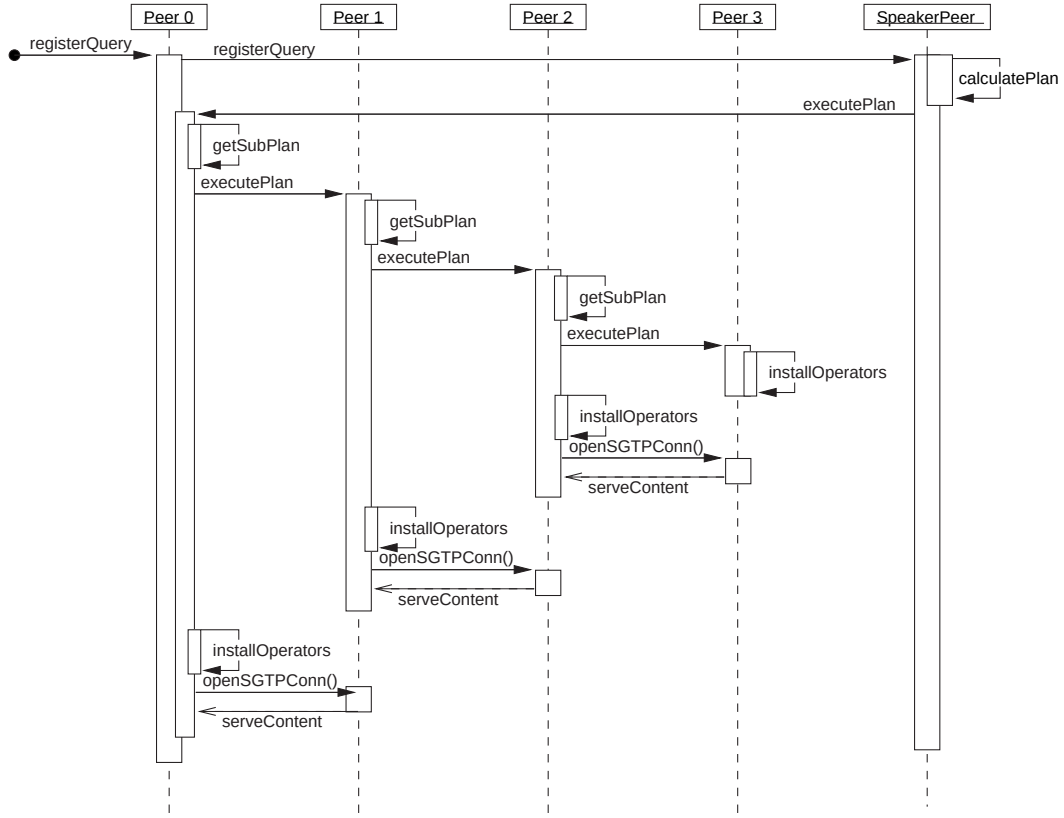


Figure 4.4: Distributed Plan Dissemination

With centralized control all processing is done in one place (the speaker-peer). In the distributed control we need to "chase around the objects" to follow the control flow. Distributed control uses polymorphism in favor to conditional logic. As polymorphism results in more methods there are more plug points for overriding and variation. In addition, effects of change are more localized.

The plan distribution processing can be done asynchronously or synchronously and there are advantages and disadvantages for both sides.

4.4.2 Synchronous vs. Asynchronous Plan Distribution

A synchronous plan distribution does not need to take multi-threading issues into account and therefore is implemented pretty straight-forward. The synchronization of the plan and its dependencies is done by the control flow. So we would install one subplan after another and finally all the local operators specified in the plan.

Executing the plans in a complete asynchronous fashion merits in more parallelism, could imitate the synchronized technique, and eventually in less temporal coupling. As a concrete example all subplans and local operators are installed in parallel.

We decided to implement a compromise between full flexibility and full synchronization. In StarGlobe first the subplans are installed and the local operators afterwards. As the local operators depend on the results of subplans, a failure during the installation process propagates through the plan hierarchy.

This moderate synchronization is also useful in the context of multi-query optimization. When a user registers a query, the subscription is forwarded to the speaker-peer, as only the speaker-peer has an overview of the streams available in its segment. Each installed query is taken into account to optimize the query execution plan for the new query. Now the speaker-peer can take advantage of the fact that its optimizer has calculated the plan. Whenever a super-peer registers a new query, successful execution only is guaranteed, if the plan was disseminated properly. Thereafter the optimizer is updated about the new query for forthcoming optimizations.

In our current prototype, however, we sometimes need to contact the speaker-peer to retrieve meta-data about the new plan and therefore the meta-data is installed beforehand. Forthcoming developers are encouraged to put all the meta-data into the plan so that optimization process and plan distribution get more decoupled.

4.4.3 Defining XQuery Plans

In the current prototype, plan installation is triggered by manually written and optimized plans. To free potentials for user-defined queries we consider providing the user with an extended WXQuery. StarGlobe could distinguish between

standard WXQuery-subscriptions and those using external operations. Then a second optimizer could be developed to perform plan calculations. Listing 4.5 gives an outlook of a possible query defining a coordinate transformation.

```
<photons>{
  for $x in stream("vela")/photons/photon
  let $r := ext:transform($x/coord/cel/ra, $x/coord/cel/dec)
  return
    <transformed>
      {$r/x}
      {$r/y}
      {$r/z}
    </transformed>}
</photons>
```

Listing 4.5: Optimizable XQuery format of external queries.

Chapter 5

System Architecture

Thin-peers are an integral part of the StarGlobe architecture. These clients provide data sources for P2P stream sharing and subscribe available data streams.

At the early stage of implementation however, they played a subordinate role for StarGlobe – and so they did in the preceding chapters. What better time than now to make a change.

5.1 Thin-Peers

For the optimization process they play a subordinate role. Their processing capabilities and their reachability cannot be foreseen and thus should not be considered by the speaker-peer when distributing the load on the network.

The speaker-peer is in charge of having an overview over the super-peer network to provide the optimizer with the necessary information. As the thin-peers are very volatile it is preferable not to burden the speaker-peer also with the maintenance of thin-peers.

We consider it as adequate to put the super-peers in charge of maintaining their thin-peers as speaker-peers are responsible for the optimization in their assigned network segment. At first sight, thin-peers providing streams do not differentiate much from thin-peers registering queries. In the complete context, however, it is arguable that these two kinds of thin-peers should be realized individually.

A thin-peer needs to register with a super-peer before it can send queries to the StarGlobe system. These "rendevouz-points" are available through a directory service or simply a web site. When it is registered, it can send subscriptions to its registrator which initiates the query installation on behalf of the thin-peer. The super-peer provides the thin-peer with a unique query-id and now the client is able to decide whether to request the actual output of the query or not.

5.1.1 Content Providers

When a stream is integrated into the system it is important for the optimizer to collect statistics about this stream. Frequency and size of building blocks are only a subset of the collected data. To support remote initialization for content providers, they have been realized as Grid services [Fen05].

A content provider is critical for queries that depend on its stream. As soon as a content provider leaves the network, queries on that data stream can no longer be processed. A query can be cancelled quite easily, unless there are different subscriptions that reuse its result stream. The system needs to assure, that reused streams are not stopped, when original subscriber has lost its interest in the query.

Only those stream operators which are running on super-peers between the subscribing thin-peer and the first super-peer where the stream is reused, can be stopped without side-effects. Depending on the overhead and load situation at the executing peer, it might be better to mark the operators as removeable instead of stopping them immediately.

5.1.2 Query Subscribers

During the implementation of thin-peers several constraints must be fulfilled. First, a thin-peer can only connect to exactly one super-peer. We check this constraint at creation time by providing only a constructor which has a super-peer as argument.

A second constraint is, that a peer should not be able to change its super-peer or connect to a second super-peer. By not providing access methods to the reference of its super-peer, a thin-peer cannot change the super-peer it is associated to. Finally, only thin-peers which are registered at a super-peer can register queries at it. This constraint is realized by providing a connect and disconnect interface for thin-peers at super-peers. Each time a new query is registered, the thin-peer must also provide its id and if the super-peer has not provided this id, the registration fails.

Class diagram 5.1 shows the relationship between thin-peers and the components of the StarGlobe it depends on. The diagram concentrates on methods and instance variables in Peer and PeerPort which are important for thin-peers. PeerPort is the Grid service interface of super-peers and Peer is the provider of this interface, thin-peers interact with.[1]

5.1.3 Query Results

When people register queries, they are most commonly interested in seeing the result stream. Otherwise, in the context of monitoring and evaluating StarGlobe,

[1] Grid Service interfaces correspond to WSDL portTypes.

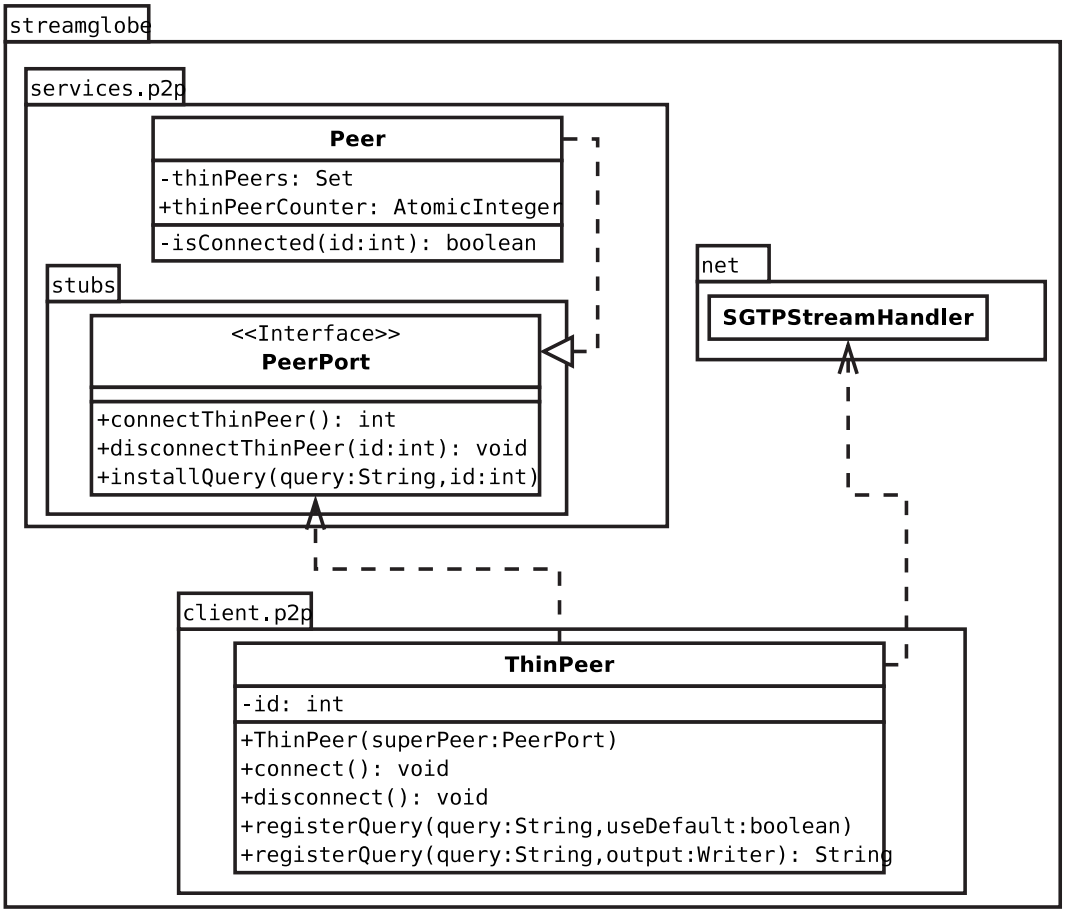


Figure 5.1: Thin-Peers and associated classes.

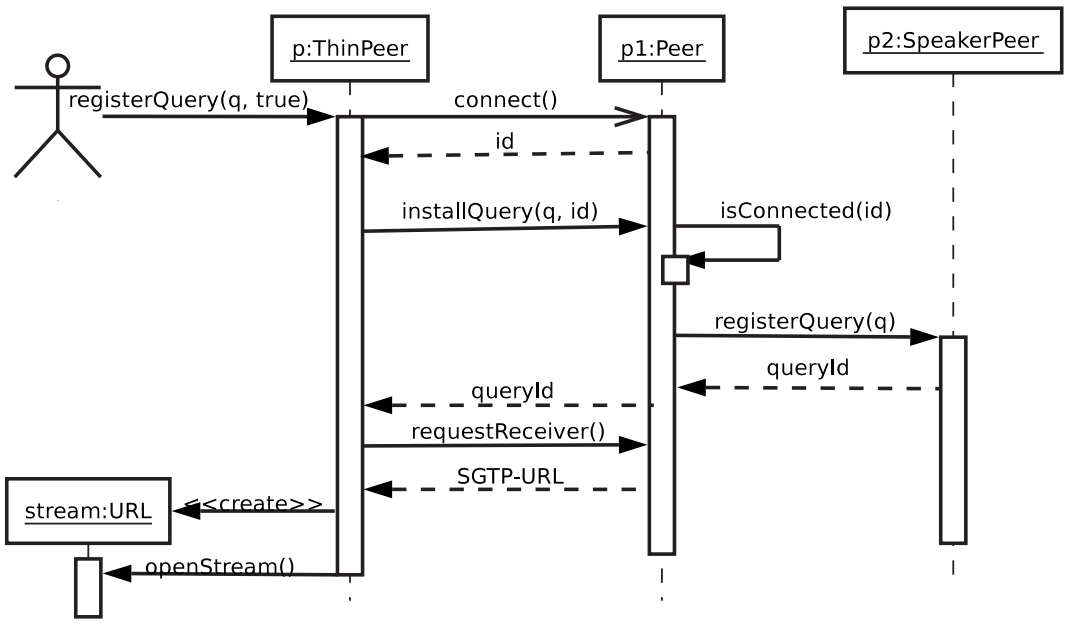


Figure 5.2: Query registration through thin-peer.

there are cases where there is no interest in the result of the query but in the impact on the overall infrastructure.

Therefore the optimization process has to be changed in a slight way. The implicit display operator which is installed at the issuing super-peer (remember that we used to register queries directly at super-peers) is now "pushed" to the thin-peer.

How can this be achieved? Either the optimizer generates display operators as well and the super peer then decides at which thin-peer the result should be displayed. An alternative solution ignores display operators in the optimization process in general. Displays are established from the thin-peer.

To display the result of a query or not can be decided dynamically.

In StarGlobe, streams are transmitted using the SGTP. When the user registers a query the system responses with the identifier of the query. Using this identifier and the connection to its super-peer, the thin-peer connects to the super-peer and displays the results.

The interaction between thin-peer and super-peer is again depicted in sequence diagram 5.2.

The implemented display used to write all data on the text console. With the more flexible approach described in this section, we can direct the output of the queries to files, to text areas in a graphical user interface or similar devices.

5.2 Generating Scenarios

A StarGlobe *scenario* can be described by the following characteristics:

- number of super-peers,
- topology of the super-peer backbone network,
- setup of queries and streams, and
- the type of the scenario (local, distributed, hybrid).

Scenarios are described with an XML document and there are several examples in the appendix to the demonstration scenarios (appendix D).

Globus-Toolkit 3 runs as container-based hosting environment – as *service container* – which is comparable to J2EE, .NET, and Sun One and provides a common framework where Grid services can be instantiated and composed [FKNT02]. Grid services can be defined as factories, meaning that several service instances can run in the same service container and interact with each other. PeerPort (the super-peer Grid service) is such a factory.

This offers a great flexibility in terms of how many super-peers run on a single computer, in a single service container, or even in a combination of those.

In the *local scenario* all super-peer instances are run on a single computer and in a single container. This setup is applicable during development or demonstrations of the system. Peers are instantiated by the *PeerFactory* of the container and reside in the same service container as the factory. So the whole network is created on a single machine if all Peers are instantiated from the same factory. To achieve a global identification (Grid Service Handle, GSH) for Grid service instances, these are generated by appending the factory-GSH with a hash value.

Though this scenario is very useful for implementation and demonstrating purposes, it is not very distributed. In a real *distributed scenario*, it is preferable to have a single Peer per station. So how can we achieve this, without changing the instantiation process? We only create one Peer from each factory.

It might be in the interest of the community if some participants run several StarGlobe peers on a powerful computer, but maybe not all have this capacity and can only one StarGlobe instance. If such a scenario is intended, we can specify a mapping between super-peers and PeerFactories and thus physical nodes.

5.3 Plan Installation

To install execution plans into StarGlobe, we provide a command-line based program, the streamglobe.client.p2p.PlanDemoClient. A example invocation for the A-Star plan would be:

```
java streamglobe.client.p2p.PlanDemoClient -p generated-gsh.txt -i 3 astarworkflow.xml
```

After starting the Globus Toolkit 3 service container, the scenario can be set up with the streamglobe.client.p2p.GridServiceGenerator. As stated above the GSHs of the individual peers change everytime, so it is important to run the GridServiceGenerator with the -g option. This stores the generated GSHs in a file.

If the plans passed to the PlanDemoClient contain the pattern @Peer$n$-GSH@, where n is a zero-based index, this pattern is replaced by the n-th line of the handle-file. The option -i m configures the client to send the plan to m-th peer (default, first GSH in the handle file).

5.4 Monitoring enabled

As our application is implemented in Java there are no platform-specific issues as long as we do not leave the virtual machine. Unfortunately the monitoring mechanisms of the Java Runtime Environment offer not very much support for tasks such as measuring the CPU load of a machine or traffic on network links. At the moment this limits us to monitor real data on Linux servers only.

On the other hand, the *statistiX* component gathers data about peers and the network load. This data then influences the result of the cost function of the cost-based optimization strategy. If the statistics show, that a network connection will be overloaded, when a candidate-plan[2] is installed, the optimizer will try to find a better query evaluation plan [Fen05].

To keep the statistics up-to-date, the optimizer integrates *Counter* stream processors in the plan, between each plan and the subplan it depends on. These processors operate on the frequency and the average size of data items, a base-load of operators and a performance-index of super-peers. Base-load and performance-index are measures designed to differentiate between operators and the processing capabilities of individual peers.

These statistics are not necessary for the other optimization strategies, as these do not consider network traffic or peer load in their calculations and therefore just would squander resources.

However, this update-mechanism to collate the statistical information, would suffice to monitor a simulated system regardless of optimization strategy. Additionally, other mechanisms (using the proc-file system, or a Java Native Interface) may come short in providing such an accurate figure, especially if "network usage" is concerned, as the Counter operators only operate on unbiased data.

By setting the boolean parameter ("monitored") to true, the optimizer can be configured to include these operators regardless of optimization strategy or platform. This enables monitoring of every StarGlobe installation event those on a single computer.

[2] A plan, the optimizer calculated to retrieve the best plan according to its cost function.

Chapter 6

Monitoring and Evaluating StarGlobe

When subscribing to a data stream via submitting a query or a query plan to our system the user is content when getting its results. From an (software-)architects perspective *this* is not enough. As described above, the StarGlobe system has a variety of configuration options and the effect of such configurations on the overall performance is important.

At the moment, command-line based clients exist to install query evaluation plans or register WXQuery subscriptions to the system. Franz Häuslschmid and Bo Feng conducted various benchmarks which supported the contention that using StarGlobe lead to a reduced load on the network and the individual peers.

Eventually, the successful application for a demonstration at the VLDB Conference 2005, was an additional driving factor to develop an interface to monitor and evaluate the StreamGlobe resp. StarGlobe infrastructure [KSKR05].

As soon as it comes to graphical user interfaces, there is an almost infinite source of ideas and neither a less amount of personal preferences which can be built into such. Therefore developing a GUI is cursed to be an "endless story" and certainly is not finished by the submission of this thesis. In the following we use the terms GUI or SGG[1] as synonyms for the graphical user interface described in this chapter.

6.1 General Requirements

Providing a single point of control is the key to monitor a distributed architecture. The flexibility of how StarGlobe can be set up enforces a clear abstraction from whether the monitored instance is running on the same or a different machine.

[1]Depending on the point of view either *StreamGlobeGui* or *StarGlobeGui*. In the implementation we used the class name StreamGlobeGui and therefore use it throughout the text as well.

People behind a SGG should be able to switch between instances during runtime and get several details on the system and its components.

One first requirement is the visualization of the *network topology*. Without SGG, the network topology only could be derived from the scenario file injected by the *GridServiceGenerator*. The development of a separate layout engine was not feasible due to resource constraints. However, the flexibility to use different layout engines is beneficial if one engine offers a layout superior to that of an other one.

An additional requirement is to display *detailed information about system components*. Several data about the system including setup, super-peers, thin-peers, connections, queries, and streams.

Setup-specific data is stored at the speaker-peer of each StarGlobe subnet and reveals information about the strategy used by the optimizer, how many queries were registered at the speaker-peer and how many of them reused preexisting streams or were rejected. Statistics about the subnet supervised by the speaker-peer (e. g., number of super-peers, number of queries) could also be displayed.

Users might be interested in the current CPU load, the upper limit of computational resources, the installed stream processors on an individual super-peer as well as which streams (or queries) are available.

Displaying the traffic for every single direction of a network link does not only offer information about potentially overloaded edges but also demonstrates how data disseminates throughout the network.

When dealing with queries, the structural information before and after processing, and thereby involved peers (subscribing thin-peer and processing super-peers) are in the interest of users and developers. These considerations map equivalently on data streams.

6.2 Demonstration-Specific Conditions

The crucible by which a successful demonstration of an infrastructure prototype will be judged is how it demonstrates benefits in an intuitive way and how it fulfils the expectations raised by the proposal. And at best, all within less than three minutes.

The main focus of the GUI clearly is the monitoring aspect and the visualization of how the load on specific network links or peers in the network is reduced by the optimization strategy.

This is achieved by showing several scenarios each setting the focus on different characteristics of the system.

Bo Feng covers details and issues about generating query-batches for benchmarks or exemplary scenarios, so we will not go into great detail on that topic [Fen05]. We developed a generator, which uses several query templates for various streams and builds scenario descriptions according to its parameters. These

scenarios vary in topology of the super-peer backbone network, in the number of super-peers, and the setup of queries and streams.

Offering various scenarios or topologies shows the flexibility of our infrastructure and gives some options to the presenters to meet different needs of individual visitors.[2] The main aspects to be delivered by the scenarios are *optimizations* gained by using data *stream sharing* and *prevention of overload* situations as well on overlay network links as on individual peers.

The demonstration setup should enable the presenter to compare different strategies. The number of parallel running instances should be enough to compare all available optimization strategies. In addition to that, several setups reduce the time needed to start a new example, since one basic example can be up and running in the background.

The *Vela Scenario* is used in other publications on the StreamGlobe system [KSH+04, KSK05]. Consequently, some attendants of the demonstration might know it. Besides being based on real (astrophysical) data from the ROSAT All Sky Survey (RASS) catalog, it is a very concrete example and therefore the queries can be referenced by names like "the Vela query" or "the RXJ query" instead of "query-0" and "query-1" on the RASS catalog (instead of "on stream-0"). A concrete example also might be easier to understand than a more abstract one.

Throughput measurements show the long-term benefits of P2P stream sharing. A result like *"x of n queries failed due to an overload situation"* with the traditional approach in contrast to a successful subscription of all queries in the optimized case should convince the critics.

Additionally, the demonstration prototype should convey where data stream processing takes place and how it actually works.

6.3 The Vela-Scenario

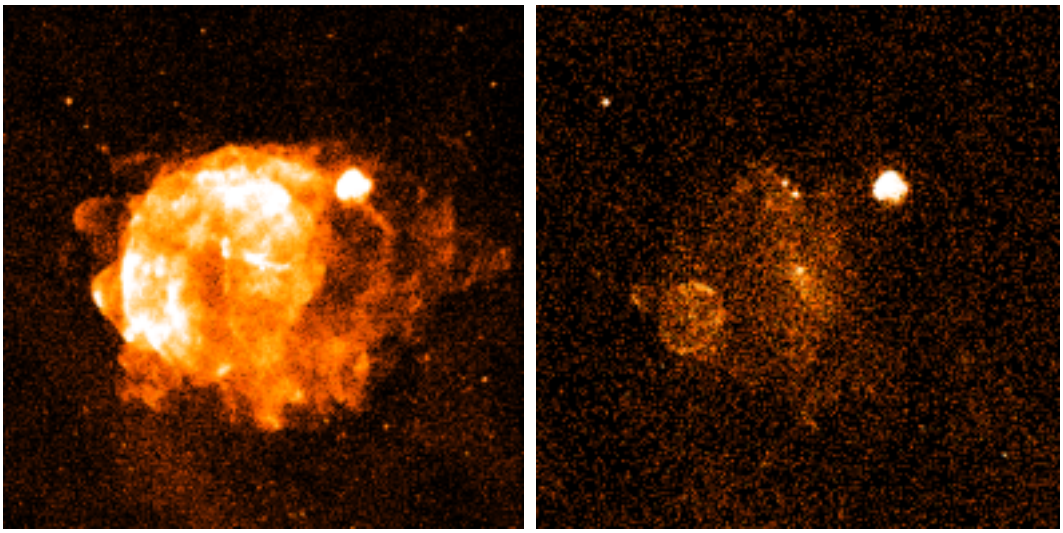
The Vela Supernova Remnant (shown in figure 6.1(a)) is visible remainder of a between 12 000 and 30 000 years old supernova in the Vela[3] constellation. The RX J0852.0-4622 supernova remnant has been discovered by observations of the ROSAT satellite [Asc98].

The MPE has provided a subset of the ROSAT catalog, which is submitted into the StarGlobe architecture. Each record of the data set resembles a photon and the XML (structured as shown in listing 6.1. It contains information about the celestial coordinates (ra, dec), information about the detection time and coordinates (dx, dy, det-time, the detection pulse (phc) and the therefrom calculated energy (en) of the photons.

```
<!ELEMENT photons (photon)* >
<!ELEMENT photon (coord, phc, en, det-time)>
```

[2] Presenters like a bit of variety, too.

[3] from *Latin* velum: sails [Wik05a]



(a) The Vela Supernova Remnant.

(b) The RX J0852.0-4622 Supernova Remnant (disc-shaped emission in the lower left).

Figure 6.1: Two ROSAT X-ray images of the same sky region in the Vela constellation taken at different energies (the second only shows photons having an energy greater than 1.3 keV).

```
<!ELEMENT coord (cel, det)>
<!ELEMENT cel (ra, dec)>
<!ELEMENT ra (#PCDATA)>
<!ELEMENT dec (#PCDATA)>
<!ELEMENT det (dx, dy)>
<!ELEMENT dx (#PCDATA)>
<!ELEMENT dy (#PCDATA)>
<!ELEMENT phc (#PCDATA)>
<!ELEMENT en (#PCDATA)>
<!ELEMENT det-time (#PCDATA)>
```

Listing 6.1: DTD of the Vela stream.

The Vela Query The first query selects photons from the area of the Vela supernova remnant.

```
<photons>
{
  for $p in stream("stream-0")/photons/photon
  where $p/coord/cel/ra >= 120.0
    and $p/coord/cel/ra <= 138.0
    and $p/coord/cel/dec >= -49.0
    and $p/coord/cel/dec <= -40.0
  return
    <vela_photon>
      {$p/coord/cel/ra} {$p/coord/cel/dec}
      {$p/phc} {$p/en} {$p/det-time}
    </vela_photon>
}
</photons>
```

The RXJ Query The second query selects photons only with a high energy impulse (greater than 1.3 keV) in the area of the RX J0852.0-4622 supernova remnant.

```
<photons>
{
  for $p in stream("stream-0")/photons/photon
  where $p/en >= 1.3
    and $p/coord/cel/ra >= 130.5
    and $p/coord/cel/ra <= 135.5
    and $p/coord/cel/dec >= -48.0
    and $p/coord/cel/dec <= -45.0
  return
    <rxj_photon>
      {$p/coord/cel/ra} {$p/coord/cel/dec}
      {$p/en} {$p/det-time}
    </rxj_photon>
}
</photons>
```

Window-Based Aggregation RXJ Query I The third query performs a window-based aggregation by calculating the average energy pulse from photons in the RXJ area. The aggregate is calculated over 20 photons, and re-evaluated after 10 new photons have arrived.

```
<photons>
{
  for $w in stream("stream-0")/photons/photon
    [en >= 1.3
    and coord/cel/ra >= 130.5
    and coord/cel/ra <= 135.5
    and coord/cel/dec >= -48.0
    and coord/cel/dec <= -45.0]
    | count 20 step 10 |
  let $a := avg($w/photon/en)
  return
    <avg_en>
      {$a}
    </avg_en>
}
</photons>
```

Window-based Aggregation RXJ Query II The final query performs the same aggregation as the previous query, however using a different window specification.

```
<photons>
{
  for $w in stream("stream-0")/photons/photon
    [en >= 1.3
    and coord/cel/ra >= 130.5
    and coord/cel/ra <= 135.5
    and coord/cel/dec >= -48.0
    and coord/cel/dec <= -45.0]
    | count 60 step 40 |
  let $a := avg($w/photon/en)
  return
    <avg_en>
      {$a}
    </avg_en>
}
</photons>
```

The four queries are installed in the order as described into the system at different peers. Some reader may already have recognized, that each of the RXJ queries can reuse the previously installed queries (as RXJ is a sub area of Vela, and non-aggregated RXJ is reusable for aggregation queries) to calculate its result.

More on the WXQuery specification and on the techniques StarGlobe uses to solve the query-containment issues arising in this context can be found in a technical report [KSK05].

Catalog	Total	Vela Query	RXJ Query
RASS Fragment	25 617 873	2 268 074 ($\approx$ 9.5 %)	6 203 ($\approx$ 0.02 %)
Demonstration Data Set	30 000	12 000 (40.0 %)	3 000 (10.0 %)

Table 6.1: Selectivities of the Vela query (query 1) and RXJ query (query 2) in the original RASS fragment and the data set generated for demonstrations.

6.4 Demonstration Data Set

The streams in the scenarios are generated from files. From the data sources provided by MPE several subsets have been converted into XML. Thus several data sets at different scales for development, demonstration, or benchmarks are available. A very fast and to some extend reasonable approach is using the first n rows of the original data. However, these turned out to be not appropriate to supply useful data for all scenarios.

Especially the construction of the data set for the paper-scenario was quite challenging on its own. The original subset (provided as CSV-file) of the ROSAT All Sky Survey (RASS) catalog contains 25617873 entries. Well, how many qualify for the Vela-query and the RXJ query? Knowing the selectivities, how are they changed?

At this point, we decide to load the data into a database, as *"select count(*) from ..."* queries on the one hand retrieve the selectivity of an individual query quite fast and on the other hand the import and export capabilities of the database are helpful in generating the final data set. Table 6.1 shows the selectivities of the first both queries in the Vela-Scenario. In the original fragment RXJ-photons make up 0.3 % of those matched by the Vela query. Assuming, an average visitor of the demo spends around 5 minutes in front of it, it becomes evident that we have to modify these selectivities. Otherwise the results for the RXJ query and the both aggregation queries would be hardly visible.

The final data set consisting of 30000 photons is generated by combining database views, load- and export-commands of DB2 Universal Database V8.2[4] and the command-line tool *sed*, a streaming editor.[5] 3000 photons qualify for the RXJ query and thus for the aggregates, additional 9000 photons are in the Vela area yet not in the RXJ area, and the remainder is outside of the Vela. The spherical coordinates of the demonstration data set are plotted in Figure 6.2 and the three-dimensional plot of 6.3 shows the energy of the photons.

[4] http://www-306.ibm.com/software/data/db2/

[5] Since SQL does not guarantee any order on the result relation besides one specifies an *order by* clause and there is no way in telling SQL to retrieve "every n^{th} data item", we combined state-of-the-art database technology with "the Power of Command Shells" [HT99].

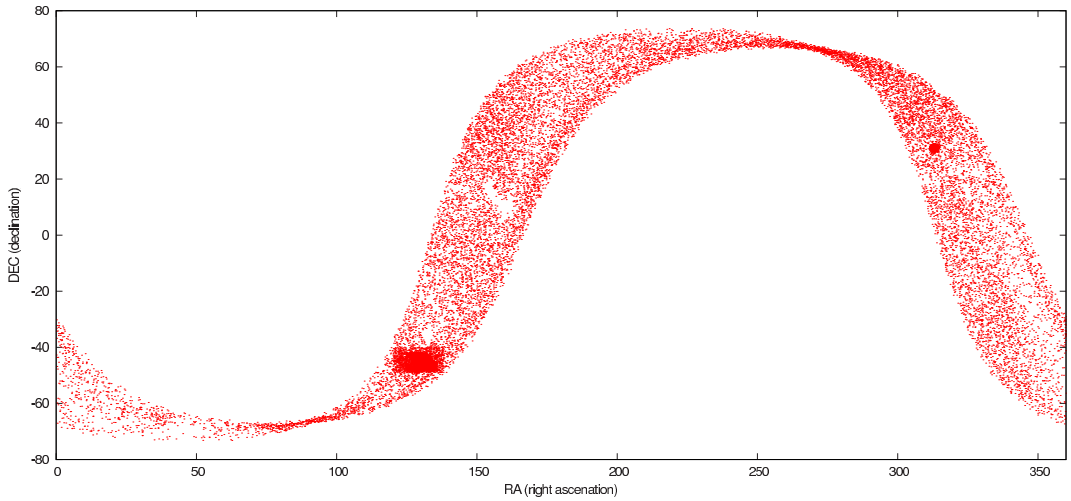


Figure 6.2: The demonstration data set. The red rectangle in the left lower part of the plot is the Vela supernova remnant. The red spot in the upper right part of the data stream is 1RXS J204813.3+332626 (alias A-Star).

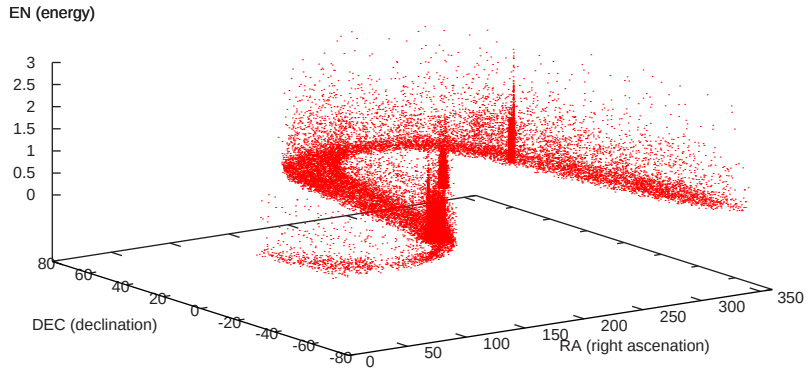


Figure 6.3: Energy distribution of the demonstration data set.

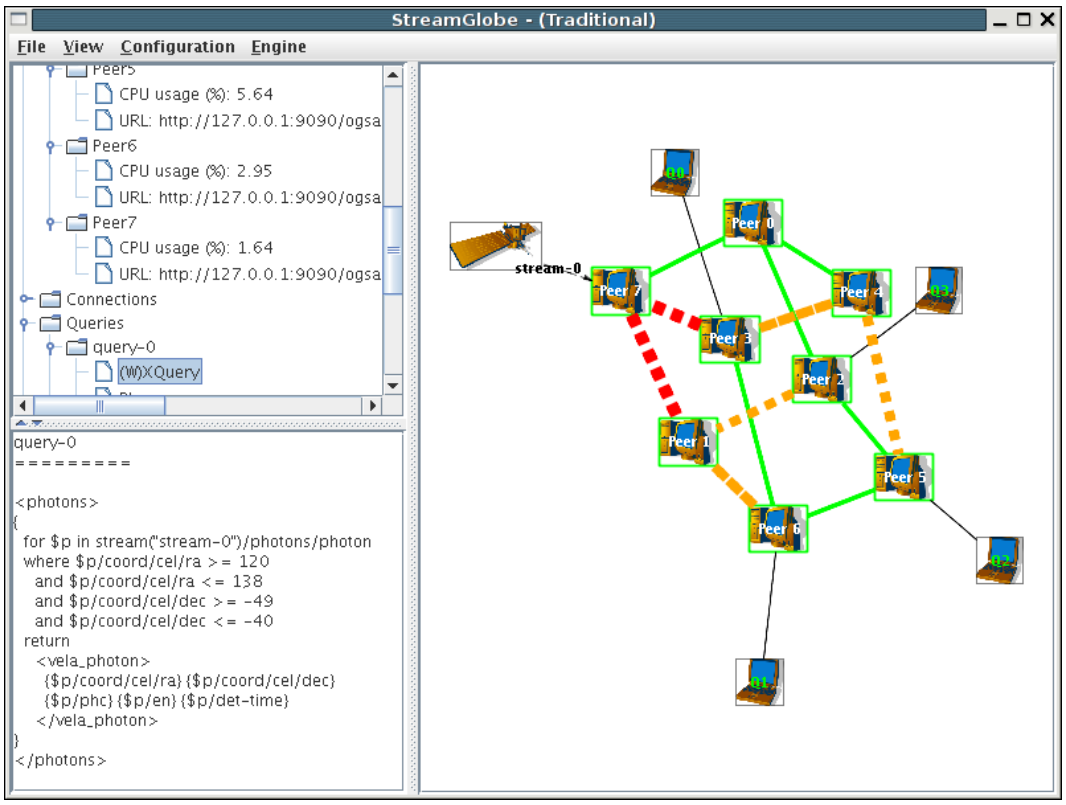


Figure 6.4: Main components of the SGG. It contains three main areas: (1) a tree view, (2) an area with detailed information, and (3) a visualization of the network topology.

6.5 SGG Design

The SGG (as shown in figure 6.4) has three main components to address the requirements presented in the previous sections. The tree view, shows the components of the visualized network (peers, connections, queries, and streams). Tree and network are in a general sense multiple views of the same data model.

The following data is shown in the tree view and the details area respectively:

Peer CPU load (measured in percent of the provided resources), URL (the GSH of the super-peer).

Connection total traffic, traffic for each direction (all measured in bits-per-second).

Query (W)XQuery source, query execution plan (both shown in details area)

Stream DTD (shown in details area)

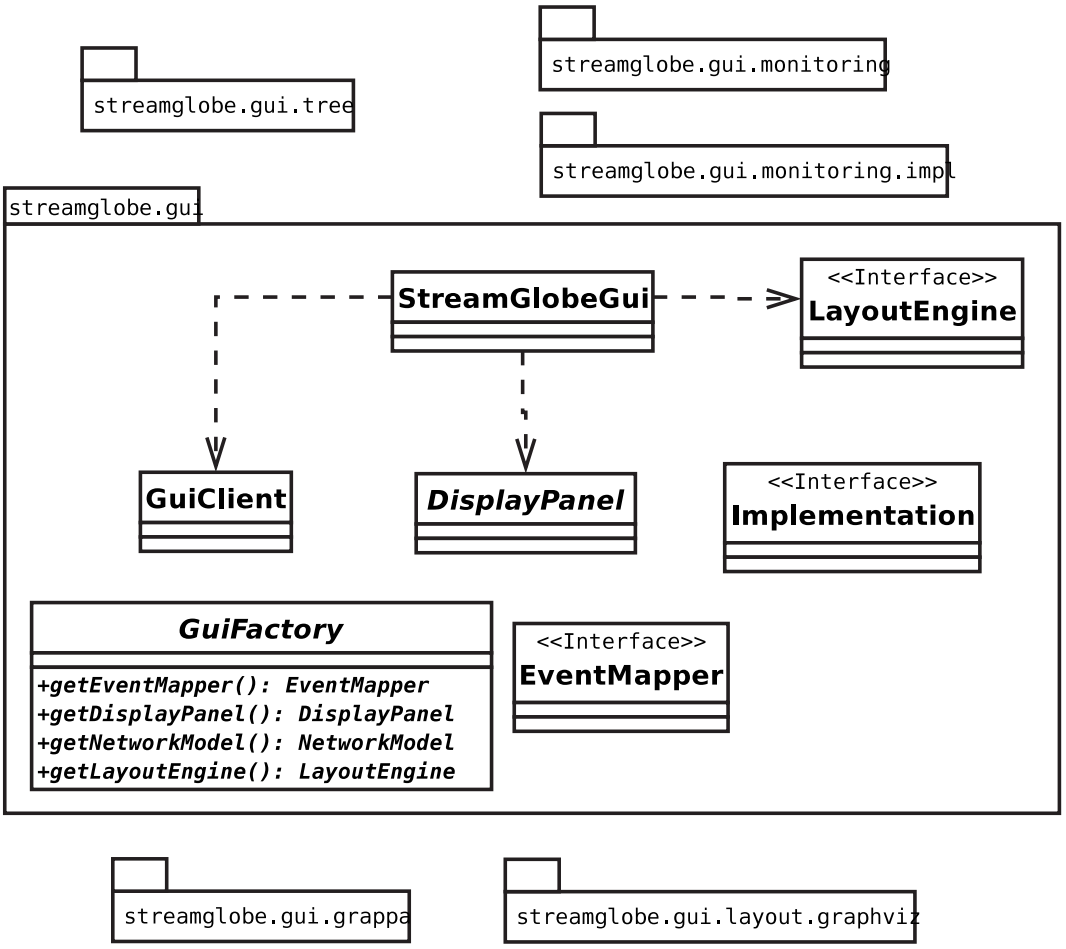


Figure 6.5: streamglobe.gui package overview

Details about the visualization of the network topology and color scheme is described in section 6.6.

The general structure is realized as shown in figure 6.5. The application class StreamGlobeGui interacts with several interfaces and abstract classes. A LayoutEngine provides the functionality to layout a network according a graph drawing algorithm, which and displayed in the DisplayPanel.

The GuiClient connects to the StarGlobe system and collects data from the system. Which kind of connection strategies are available, an overview of the both associated packages, and how connections are set up is described in the next section.

The EventMapper interface provides methods to map selection events of the tree view and the implementation of the network visualization. The components of a network are subsumed in the streamglobe.gui.model package, where abstract classes provide access to the concrete implementations via the Implementation interface. That enables the event mapping mechanism to access concrete implementations, without the need for a more specific interface.

To keep the system independent from the layout engine and the display component, we use the *Abstract Factory* design pattern [GRJV95]. The GuiFactory provides the SGG with implementations of the interfaces described above.

GraphViz[6] is our layout engine of choice, as the project provides a variety of different layout algorithms. Additionally, *Grappa*[7] is a Java library to display such graphs embedded in Swing. GraphViz uses the *Dot* grammar to describe graphs . To implement the basic interfaces SGG is depending on based on the Grappa library , is the task for streamglobe.gui.grappa. Before we outlay the structure of the streamglobe.gui.layout.graphviz module we cover how the actual monitoring mechanism works.

6.5.1 StarGlobe Monitors

The StarGlobe system can be monitored from a graphical user interface or just a command line tool. However, there are common needs of both components that are extracted to the MonitoringClient class in the streamglobe.gui.monitoring package (see figure 6.6). When users are only interested in the topology it is appropriate only to get updates about topology changes as peer addition or removal. This can be triggered with the collectsDetails flag. Moreover, if an interesting snapshot is currently displayed in a visual tool (e. g., during a demonstration) a mechanism to control the monitoring activity is necessary. As speaker peers control the segments of the StarGlobe network, these are connected to get the basic topology of their subnet. Thus access to the other super-peers in the subnet is achieved. How the data exchange between monitor and system is driven can

[6] http://www.graphviz.org/

[7] http://www.research.att.com/ john/Grappa/

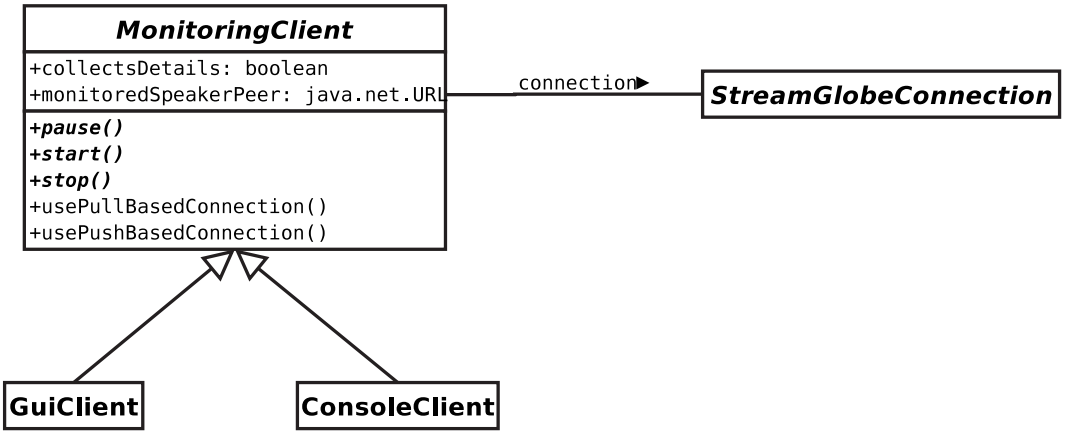


Figure 6.6: Monitors for the StarGlobe system.

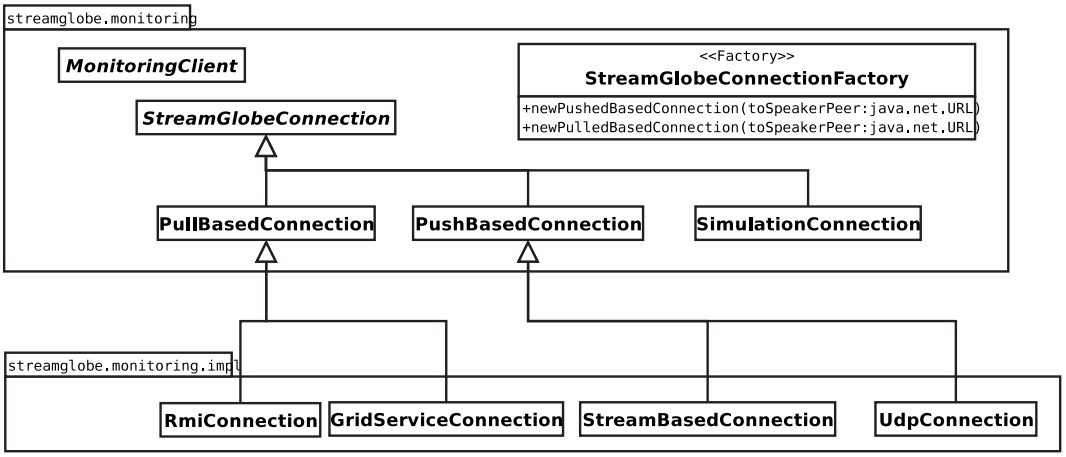


Figure 6.7: Connection strategies for monitors.

be addressed in various ways.

Communication method

StarGlobe provides Grid services especially designed for monitors to retrieve statistics and data about the current state. Using these services keeps the system in the paradigm and reuses already implemented modules. On the other hand this approach does not scale for a large amount of peers due to the enormous overhead using SOAP-Messages. Using a non-XML based format (e. g., Remote Method Invocation) could reduce the message size. The *pull-based* approach is well-suited for small scenarios yet reaches the border of its efficiency with increasing topology size.

Another group of communication methods can be subsumed under *push-based connections*. As the information displayed at the GUI is quite volatile it would

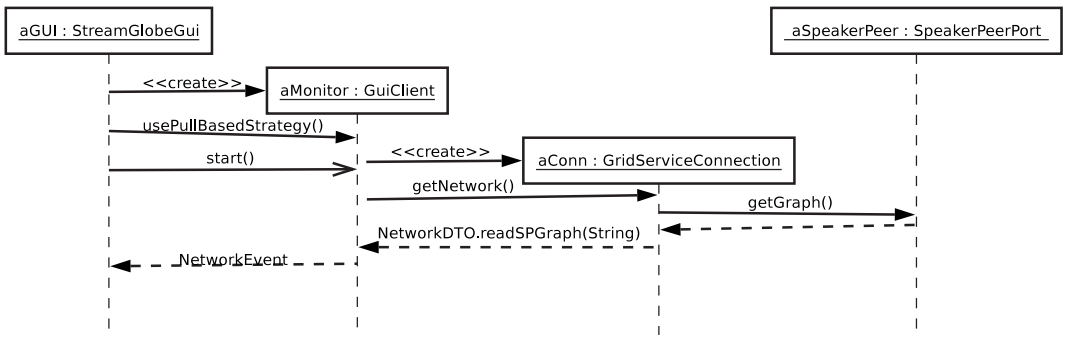


Figure 6.8: Sequence diagram of a (pull-based) connection setup between SGG and StarGlobe.

be possible to use UDP datagram communication to send updates to SGG.

An appealing alternative could be to register the GUI as a thin-peer at the speaker-peer which then multiplexes the information gathered from the super-peers into an event-stream. At the GUI end of the communication these different approaches are hidden behind an event-based interface.

The StreamGlobeConnecitonFactory is suitable for cases whenever the access method of StarGlobe with a push-based or a pull-based approach respectively is used without relying on a specific implementation.

Figure 6.7 sums up the different strategies and also indicates that for testing purposes a SimulationConnection can be implemented as well. The current prototype only implements the pull-based approach using Grid services.

Connection Setup

To connect to the running StarGlobe instance, the user specifies a Grid service handle of the SpeakerPeer service. Then the GuiClient is started, which uses an instance of ConnectionStrategy it was configured to use. The approach described in figure 6.8 connects with a configurable frequency the StarGlobe system to retrieve the current network topology and further information about the monitored components. The communication between MonitoringClient and SGG is event driven.

6.5.2 Layout Engines

Having received such a network event, it must be annotated with positioning data by the LayoutEngine. The only method in the interface converts a NetworkDTO into an appropriate format, annotates it with the layout information and converts it back to the *Data Transfer Object (DTO)* [Fow02]. In this particular example it increases the flexibility of our GUI. At the moment DisplayPanel implementation and LayoutEngine implementation use the same graph representation. The introduction of the DataTransferObject now enables us to use a GraphML based

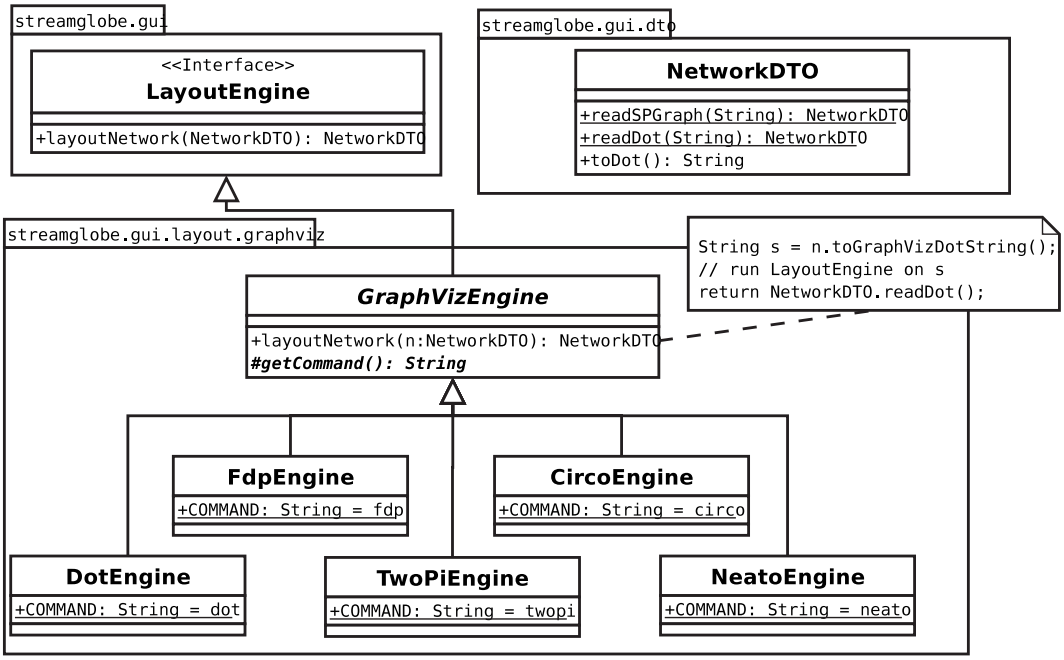


Figure 6.9: Overview diagram of the streamglobe.gui.layout.graphviz package.

LayoutEngine by simply adding a static NetworkDTO readGraphML(Element) and a Element toGraphML() method to NetworkDTO and leaving the remaining system unchanged.

Besides the classes involved into the layout process, figure 6.9 shows how the different GraphViz layout algorithms are integrated. *Dot* draws graphs hierarchical, *neato* and *fdp* use variants of spring models, *twopi* uses radial layout, and *circo* draws graphs in a circular layout.[8]

6.5.3 StarGlobe Data

To provide the optimizer with all the necessary data, the speaker peer gathers information about streams and queries in ReuseStreamInfo and QueryInfo objects in maps of the ReuseInfoManager.

As conceivable from the class names in figure 6.10 again the *Data Transfer Object* pattern is used. Derived from the dependencies, the loose coupling between the transfer objects and the data sources in the StarGlobe system is a valuable benefit. The locality of changes is very high, as refactorings in both domains would in only propagate to the SpeakerInfoAssembler.

[8]see *man*-page for *dot* (http://www.graphviz.org/cgi-bin/man?dot).

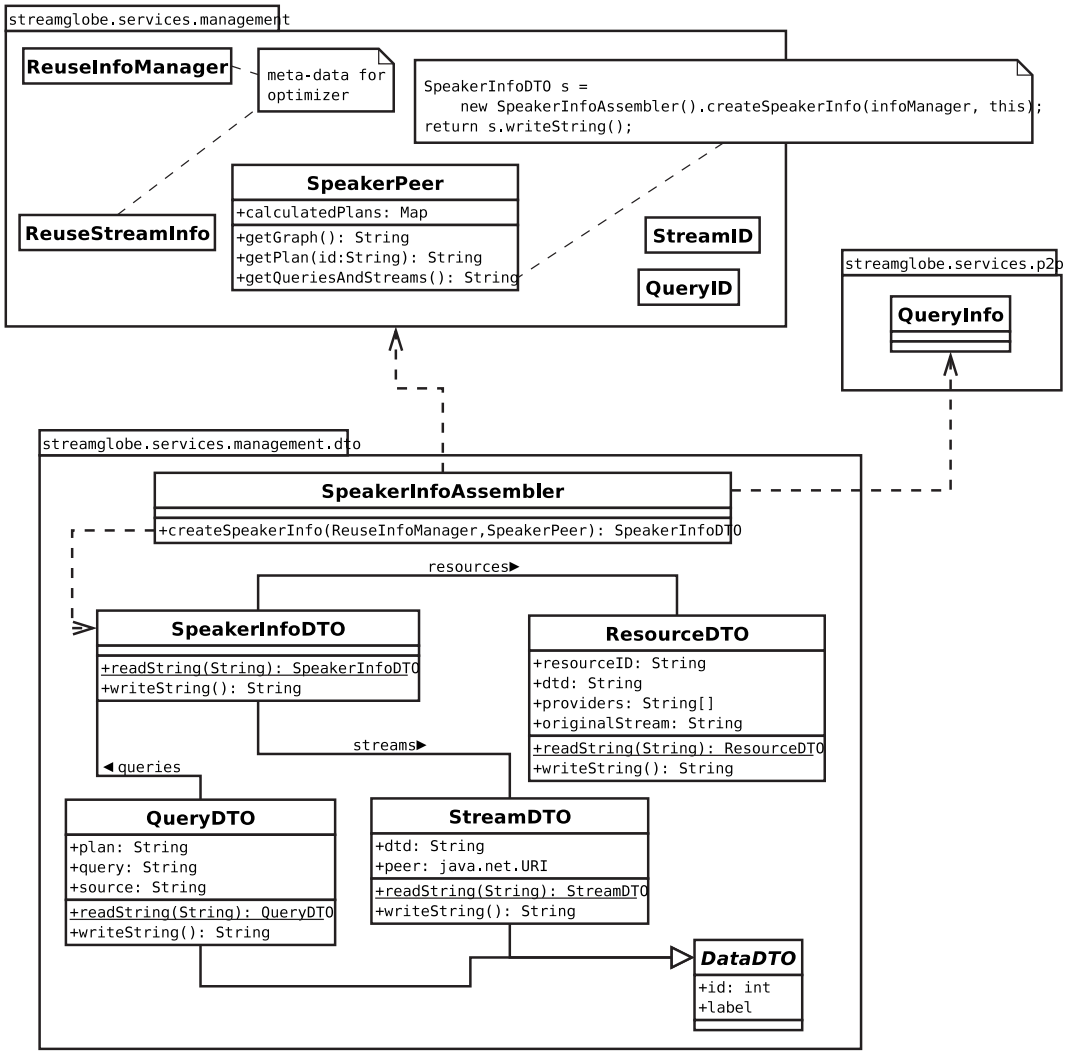


Figure 6.10: streamglobe.services.management.dto package overview

6.6 Running the Demo

The individual setups to compare different strategies, network-topologies or throughput scenarios differ in some important features. Each service container (an instance of Globus Toolkit) uses a different service-port for the grid services and an individual setup for the discovery mechanism of StarGlobe. In addition to that, its setup contains configuration parameters for content providing thin peers.

Since at least some of these parameters are materialized in the scenario description for a StarGlobe setup, it seems appropriate to generate all final scenarios from a template using the individual setup characteristics. These characteristics are stored in property-files to enable developers or presenters to adapt those fields according to their computer.

In order to decouple the used strategy from the individual setup strategy-specific parameters are stored in a separate property-file. Which strategy to run at which installation can be decided at start-up time.

This flexibility is achieved by using Apache Ant, a Java-based build tool [Fou05]. Especially the fact that no environment variables or classpaths need to be specified (and we certainly would forget them on the next installation) is another incentive.

Some scenarios are predefined in the Ant script. In general, Ant expects runtime parameters defined as properties which might be not very convenient for users non-familiar with Ant. We therefore provide Perl-script runDemo.pl, which moreover enables us to present different setups than the predefined ones.

In the following, we present the course of events during the Vela scenario when using the conventional strategy and the cost-based optimizer respectively.

6.6.1 Vela Walkthrough

Starting Point

Both strategies have the same starting point (figure 6.11), a hypercube topology of eight super peers and the Vela data stream injected at *Peer 7*. The color of the connections and the boxes around the super-peers visualizes the load of the network component. Both can vary from green over orange to red. The conventional approach is to install the queries operators at the super-peer where the querying thin-peer has registered. Content-provider are depicted with small satellites whereas thin-peers as laptop showing a shortened QueryID. If the color of the query-id is red, either the query has been registered at the super-peer but failed due to an overload situation or the plan is currently processed.

Vela Query

After the Vela Query has been installed, the traffic on the link between *Peer 7* and *Peer 3* in the conventional setup is higher than with the cost-based strategy

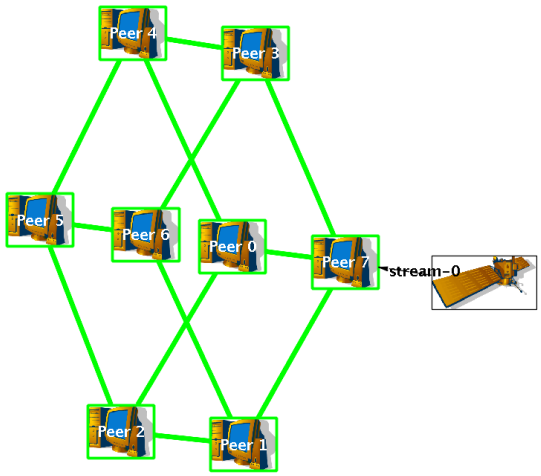


Figure 6.11: Starting point of Vela scenario.

(figure 6.12). The conventional strategy transmits the Vela stream to *Peer 3* and processes it there, whereas the cost-based optimizer installs the query at *Peer 7* and therefore only transmits data across the network that belongs to the Vela query. This avoids superfluous transmissions which is shown as less network traffic. Data transfer is visualized by dashed lines as well as by changing the line width proportional to the transmitted data stream.

RXJ Query

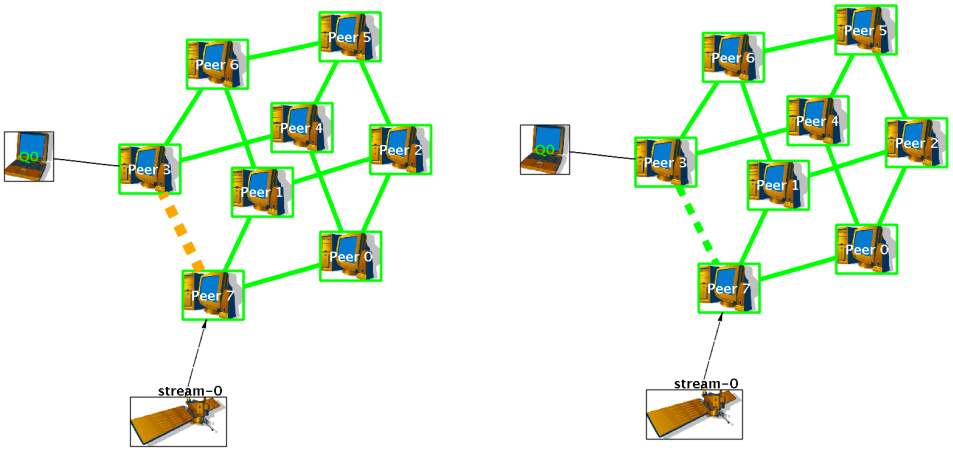
The RXJ query is installed at *Peer 6*. As this query is completely contained in the Vela query, the cost-based optimizer reuses the existing stream (figure 6.13). The optimizer installs the RXJ filter at *Peer 3* which only sends photons to its neighbor if they are in the RXJ area and have a high energy pulse. The conventional approach, instead, redundantly transmits the unfiltered ROSAT catalog to *Peer 6*.

Window-Based Aggregation RXJ Query I

With this query, the network link between *Peer 7* and *Peer 3* gets overloaded. This reinforces that the cost-based approach is superior as it reduces the network load. Besides stream sharing, the optimizer uses another optimization technique: late re-structuring. Renaming of tags (like rxj_photon) is done at the last super-peer before the final thin-peer.

Window-Based Aggregation RXJ Query II

Eventually, the last query leads to the second overloaded connection in the conventional case. Despite potentially longer installation times due to the optimiza-



(a) Conventional. (b) Cost-based.

Figure 6.12: Network after installed Vela Query

tion process, the comparison of the network topology shows the benefits of the optimized approach resulting in less peer load and less network traffic.

6.6.2 Throughput examples

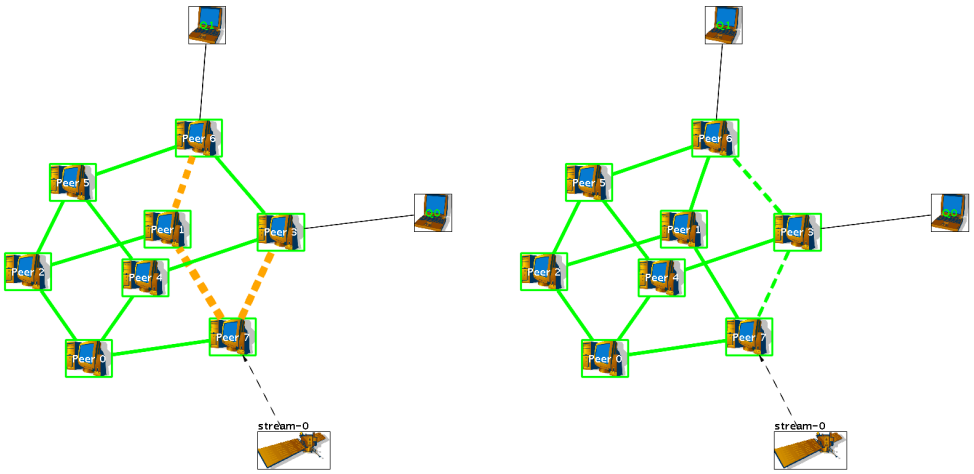
At the same setup as in the Vela scenario (eight super-peers, hypercube topology) a query batch of 25 queries is registered to show throughput measurements. The traditional approach will fail in registering several queries due to overloaded connections, the cost-based optimizer successfully installs all queries and even 19 queries can reuse pre-existing streams.

The query batch is generated with streamglobe.scenario.VelaQueryGenerator which provides templates for two window-based aggregate queries and two selection templates (on energy and on coordinates). They are combined with one of two available result templates.

Ten of these 25 queries select on the energy value, eleven queries contain a constraint on the coordinates and the remaining four queries split equally across the two window-aggregates.

6.6.3 Demonstration of Bypassing

Our cost function based optimizer can significantly reduce network traffic and balance the processing load among several super-peers. However, there are situations where even this approach runs into problems which can be circumvented. One of these occasions is an overloaded connection on a shortest path and is demonstrated in the following scenario.



(a) Conventional. (b) Cost-based.

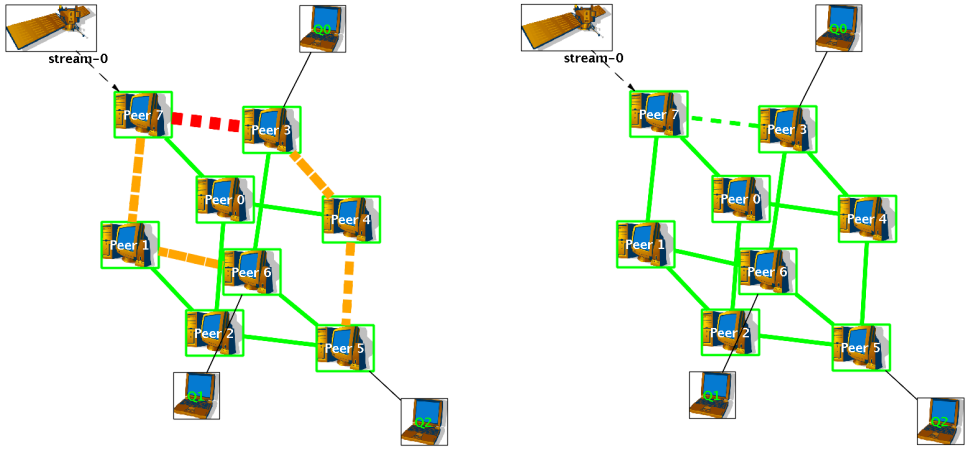
Figure 6.13: Network after registration of RXJ Query.

At super-peer $SP\,2$, two astrophysical catalogs are provided. A thin-peer registered at super-peer $SP\,0$ wants to display the first catalog. That burdens a considerable amount of traffic onto the overlay connection and no other traffic can be transmitted over this connection from now on.

When the next query gets installed at super-peer $SP\,3$ the ordinary cost-based optimizer fails, as the shortest-path between $SP\,3$ and $SP\,2$ via $SP\,0$ cannot be used due to the heavy load on the connection between the latter nodes. Stream-reuse is not possible as two different streams are queried and the optimizer fails to see the longer, yet available route along $SP\,1$. Optimizers configured to bypass overloaded connections and peers recognize this situations and remove overloaded elements (edges or peers) from the optimization graph and thus would find the route described above.

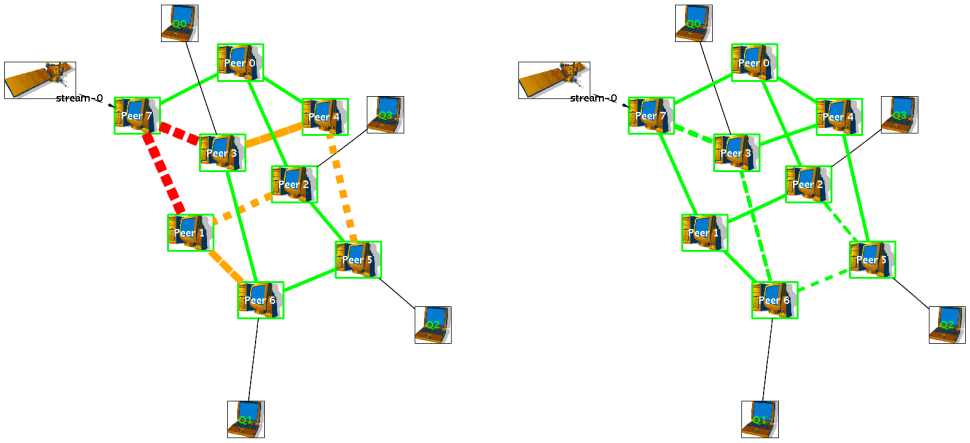
Unfortunately, this strategy was not realized until the conference, yet after implementation, it can serve for forthcoming demonstrations and support the robustness of our system.

Together with stream widening we anticipate to receive even better results.



(a) Conventional. (b) Cost-based.

Figure 6.14: Network after injection of first aggregate query.



(a) Conventional. (b) Cost-based.

Figure 6.15: Network after injection of second aggregate query.

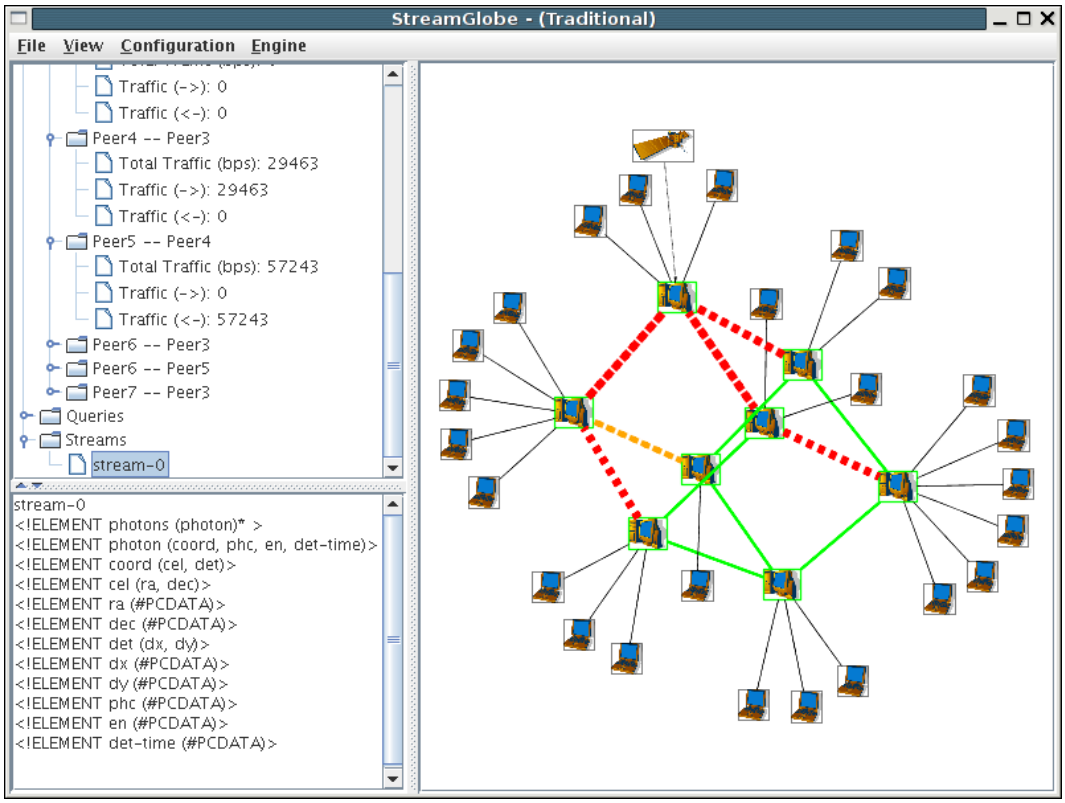


Figure 6.16: The traditional Throughput scenario.

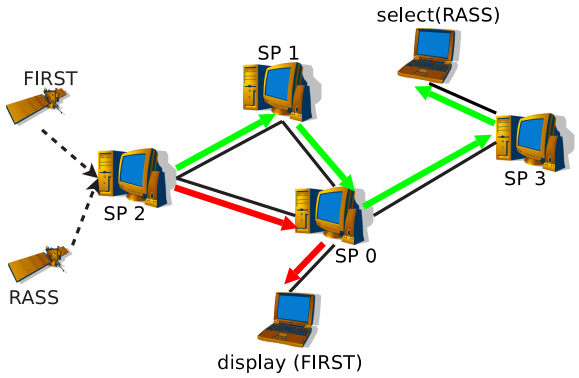


Figure 6.17: The plane cost-based optimizer would reject the selection-query due to the heavy traffic caused by the full-display query installed first. A bypassing optimizer recognizes this overload and finds a longer yet practicable route.

6.7 Evaluation Outlook

The graphical user interface visualizes the current state of our infrastructure. However, it is not conceivable how the state has changed over a longer period.

Therefore an evaluation client was developed to collect the identical data as the graphical user interface. It stores the events of peers and connections in separate files. A peer event consists of a timestamp, the peer identifier, and the current CPU load. The connection events are composed from a timestamp, both peer identifiers, and finally the overall traffic.

The first evaluations were computed over a simulated setups and it is planned to evaluate the scenarios on a real distributed setup to use the results for benchmarks.

Three dimensions are envisioned: average load comparison, registration time, and query success rate.

Figure 6.18 and figure 6.19 show the average load on the individual peers and connections respectively in the Vela scenario and the throughput scenario.

On peers the load is measured in percent of the provided CPU share, and the network traffic is plotted as bits-per-second. As expected, the traffic in the conventional case is very high on all used edges due to redundant transmissions in both scenarios. To make the results comparable, all measurements during query registration have been neglected for the average calculations.

The results of the query installation times in both scenarios and the statistics about installation and reuse in the throughput scenario are shown in the tables on page 77.

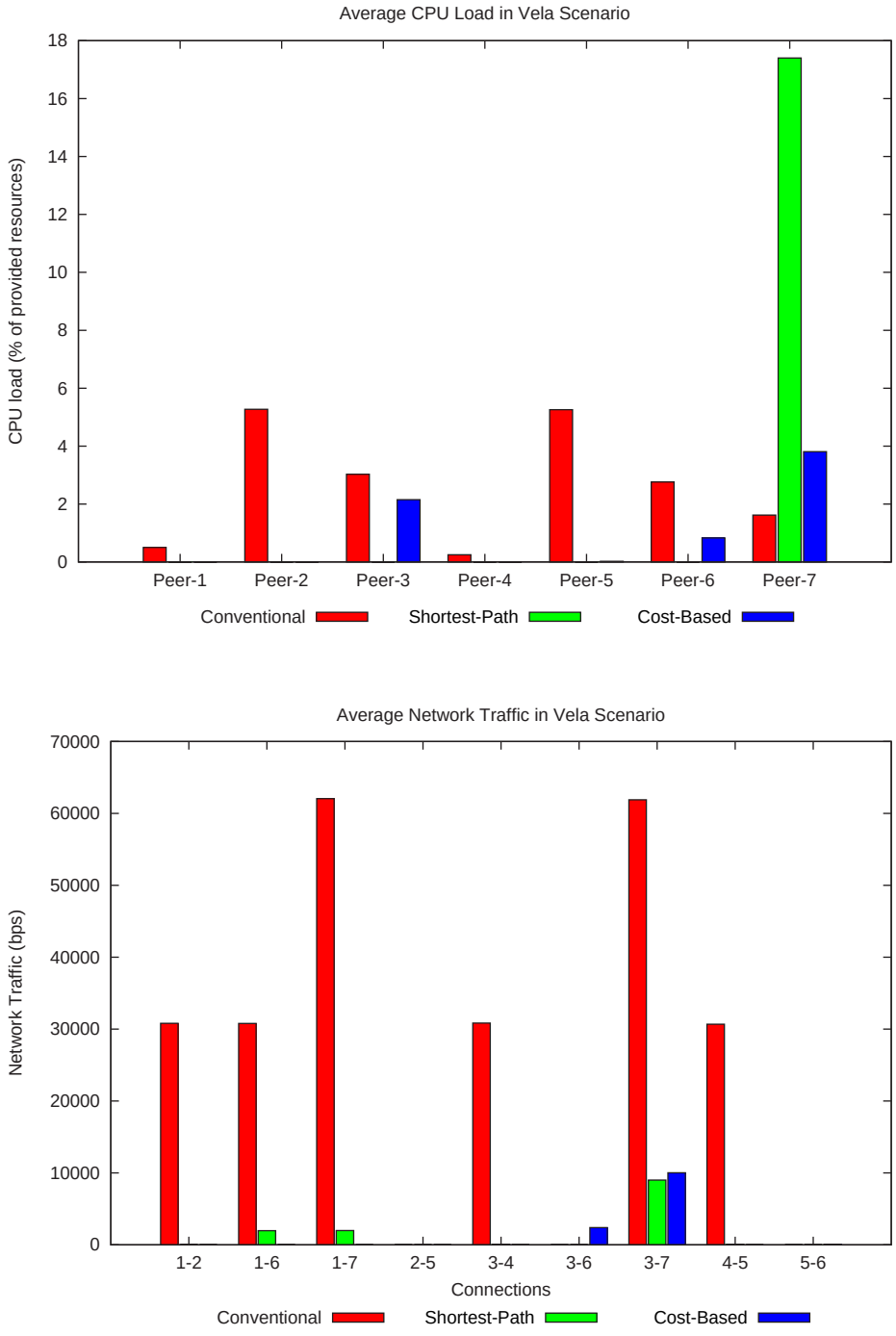


Figure 6.18: Average Load in simulated Vela Scenario.

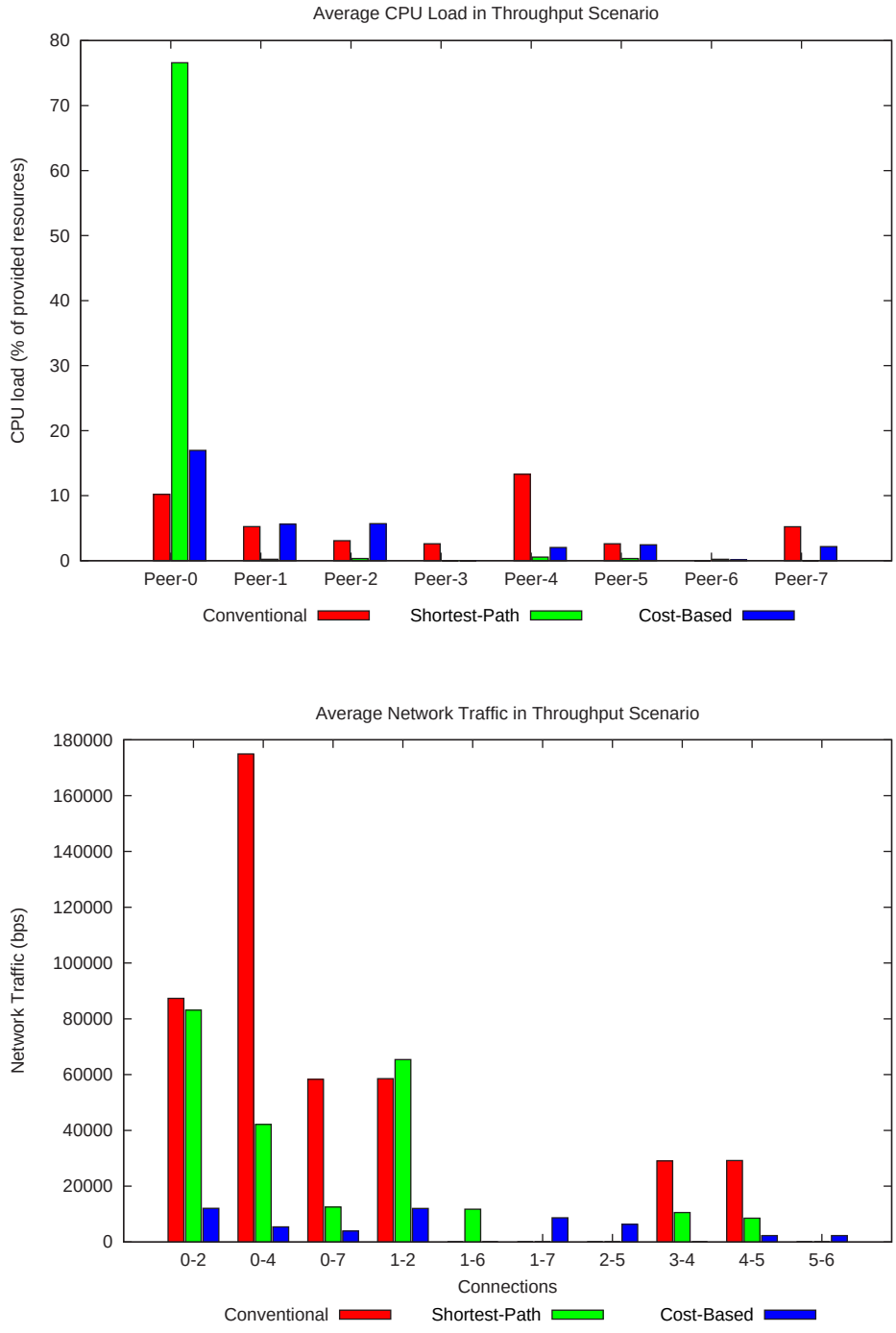


Figure 6.19: Average Load in simulated Throughput Scenario.

Registration Time (ms)	conventional	shortest-path	cost-based
Average	2599	2417	2122
Minimum	1867	1448	1685
Maximum	3855	3759	3330

Table 6.2: Query registration time comparison (Vela scenario).

Registration Time (ms)	conventional	shortest-path	cost-based
Average	1209	953	1982
Minimum	284	287	774
Maximum	2444	2372	3839

Table 6.3: Query registration time comparison (Throughput scenario).

	conventional	shortest-path	cost-based
Installed	14 (56 %)	25 (100 %)	25 (100 %)
with stream reuse	0	0	19 (76 %)
rejected	11 (44 %)	0	0

Table 6.4: Query success rates and overview (Throughput scenario).

Chapter 7

Related Work

At the beginning of this thesis its integral technologies are presented . Browsing through the proceedings of various conferences of the database or communication networks communities substantiates that "Grid Computing", "Peer-To-Peer Networks", and "Data Streams" are subjects of ongoing research at various institutions.

We cannot treat them all exhaustively and thereby choose only a selection of related research projects and differentiate them to StarGlobe. We elaborate common grounds, the focus of the related publications and what of their ideas could be interesting for adaption.

To begin with, concepts and techniques of the ObjectGlobe project have had a great influence on this thesis. As we have discussed at the beginning and in chapters 3 and 4, this thesis applies technologies of ObjectGlobe in the context of data streams [BKK+01].

7.1 Grid Computing

As StarGlobe originates from StreamGlobe it also builds on the Globus Toolkit, the reference implementation the Open Grid Services Architecture (OGSA) and augments it with data stream processing capabilities. This section introduces systems that have also aspects of grid computing on their agenda.

Another system based on the Globus Toolkit is GATES (Grid-based Adaptive Execution on Streams) [CRA04]. Liang Chen et al. also motivate their research by the increasing importance of data stream management systems in the context of increasing data volumes at distributed sites and applications like data processing from scientific instruments, computer vision based surveillance, or online network intrusion detection. They pursue four main goals with their architecture: to use existing services provided by OGSA as much as possible, support distributed processing of several data streams, enable adjustment with regards to real-time constraints, and easy deployment of applications.

This alternative approach lays its focus on adaptive queuing techniques, data stream analysis and quality-of-service aspects in data stream dissemination. As in StarGlobe users can specify XML query evaluation plans, Gates users specify configuration information of applications in XML which then can be used by others to launch the application. The idea of function repositories (where users share functionality among each other) which could be compared with this as well, lays more emphasis on the dynamic aspects and the interface for user-defined operations.

PlanetLab [Pla05] is a platform for deploying, evaluating, and accessing globally distributed network services and support research in innovative network infrastructures.[1] PlanetLab decouples the operating system from the several distributed network services as a Virtual Machine Monitor (VMM)[2] using the principle of distributed virtualization. Each service gets its individual VM (called slice) on several peers.

PlanetLab cannot be compared to the StarGlobe system itself, yet it seems like an interesting testbed for the StarGlobe infrastructure.

As a last example of related work in the field of Grid Computing serves Jalapeno, developed by Niklas Therning and Lars Bengtsson [TB05].

The Jalapeno framework aims at providing decentralized grid computing based on a P2P infrastructure to solve "embarrassingly parallel problems". With respect to a submitted *task bundle* peers are either a *task submitter*, *managers* which distribute the individual *tasks* among their subordinated *workers*. When several tasks arrive at a manager it submits some to its workers and routes the remainder to other connected managers. Undelivered tasks are returned to the task submitter and sent to a new (randomly chosen) manager.

Jalapeno focuses on grid computing as a means for solving problems too hard for a single computer by connecting the idle resources of interconnected computers. So one could see it in the tradition of the projects as SETI@home [SET05] or distributed.net [dis05]. However, it increases fault-tolerance by work stealing (i. e., tasks are retransmitted into the network after a timeout if no results are returned) and, as implemented in Java, provides heterogeneity and security (through platform independence and sandbox principle) and anonymity which is provided by the JXTA P2P framework.[3]

The concept of workers and managers is kind-of similar to the speaker-peer and super-peer ideas realized in the StarGlobe system, as managers subdivide task among their workers and speaker-peers optimize the data stream with regards to query reuse. So this project could provide interesting concepts how to organize

[1] This paragraph is adapted from an article by Anderson et al. [APST05] where PlanetLab is used as a virtual testbed for applications.

[2] a program running on a computer and virtualizing the resources so that individual services can use them.

[3] The JXTA framework [JXT05] provides a generic form of communication and services to the application programmers set atop of concepts like distributed hash tables.

the subnet structure and decide how large such subnets should be.

Leaving the context of Grid Computing, Jalapeno is actually not the only infrastructure to use the JXTA framework as we will see, when we proceed to the next section.

7.2 Peer-To-Peer Networks

The stunning success of P2P file sharing applications and the increasing share of Internet traffic which originates from these networks has led to intense research on this topic.

Various research projects have investigated in overcoming the deficiencies of unstructured P2P networks like Napster (connection setup over a centralized index) and Gnutella (message flooding) to more sophisticated alternatives using technologies like distributed hash tables. The most prominent implementations of these technique are CAN (Ratnasamy et al.), CHORD (Stoica et al.), Pastry (Rowstron et al.), and getting a increasing share BitTorrent (Cohen) [RFH+01, SMK+01, RD01, Coh03].

Another project conducted at the group of Alfons Kemper focussing on distributed query processing is QueryFlow. It focuses on optimization of distributed queries and uses a super-peer backbone network as well. QueryFlow also distinguishes between highly-available (super-)peers and peers having an intermittent connection. The routing of queries in the infrastructure is done with a set of distributed indices (SP/SP-indices, SP/P-indices) and queries are installed as abstract query plans, which then are expanded on-the-fly by the processing super-peers. The decision to delegate the maintenance of thin-peers to the super-peers (although in a different context) was also influenced by the idea of distributed indices. For more details on the query dispersion in QueryFlow we refer the interested reader to the references [KW05, DKNW04, KW01].

Besides the super-peer structure, StarGlobe also shares the hierarchical differentiation between thin-peers and super-peers with QueryFlow. The transparency of the individual small peers connected to a super-peer for other super-peers reduces the amount of information an individual peer has to cope with. To reduce the messaging overhead super-peers are arranged in so-called hypercubes using the HyperCuP-Protocol [SSDN02] and Alfons Kemper and Christian Wiesner cooperated with Ingo Brunkhorst et al. of the University of Hannover on the influences schema-based information retrieval has on query processing [BDK+03].

In several co-publications, the discussion is focused on schema-based P2P topologies to store, access, and update distributed knowledge about data distribution. They further address how this knowledge can support the distribution and expansion of abstract query plans in the topology and thus be more efficient in the usage of distributed computing resources.

For more information on the challenges and design issues of schema-based

P2P architectures the interested readers are referred to Wolfgang Nejdl et al. [NSS03], which gives an overview on the experience the authors gained from the research on the Edutella framework.

To round up the discussion on related P2P system we choose the work of Gert Brettlecker et al. from the University for Health Science, Medical Informatics and Technology (UMIT) [BSS04].

They see the incentive of P2P technologies together with data stream management systems predominantly in health care applications and focus on flexibility and reliability of distributed process management and distributed stream management.

Hyperdatabases, which address the composition of user-defined processes over existing services and provide an infrastructure for process execution, are augmented with streaming processes.

An apparent difference to StarGlobe is the explicit sequencing of stream elements and the request for redistribution if some of the packets are lost during transmission.

Especially their work on operator migration and backup concepts for the internal state of operators (which is necessary when operators are relaunched at a different site) are well-founded. Besides the options discussed in the implementation section of the plan distribution, the proposed techniques are another alternative to realize transactionality in data stream processing.

7.3 Data Streams

Data production at increasing speed faces us with the necessity to process data before we store it persistently. Then applications evolve which consider long or infinitively lasting queries and thus provide incentives to research on *Continuous Queries (CQ)*.

TelegraphCQ focuses on challenges which arise from processing continuous queries in networks with uncertainties [CCD+03]. Sirish Chandrasekaran et al. consider as uncertainties on the one hand the high volatility of network environments (e. g., sensor networks) and on the other hand interacting users can change their queries due to results they have seen so for. Such modifications are gracefully adapted by their system.

The ONYX prototype architecture as presented by Yanlei Diao et al. aims at content-based data dissemination [DRF04]. Information is delivered to users by matching the content against profiles containing the users' interests. These profiles are also described with an XQuery fragment, however to achieve the content-driven routing, query processing is done using two planes. The first monitors the data flow (data plane), whereas the second captures flow and changes of queries.

Wee Siong Ng et al. propose CQ-Buddy and approach the problem of data stream optimization from the data providers perspective [NST03]. If all query

processing is done independently the response time of the system decreases and eventually data providers are the bottle neck of the infrastructure. After providing a model for similar queries they allow query sharing on a single node as well as between nodes. In CQ-Buddy overloaded data providers can ask neighbors "for help" to act as proxies by selecting them using an adaptive strategy based on lottery scheduling techniques. In comparison, the optimization strategy of StarGlobe, is more proactive as it optimizes every query according to network-traffic and peer-load. Both systems have a notion of "weak" and "strong" peers and balancing the processing load across the network. Interesting for networks having a higher disconnectivity is the support for *pervasive continuous queries* which are long-running queries that are calculated on behalf of the registering peer (which meanwhile might have gone offline) and whose results are stored at the calculating peer for later retrieval of the original subscriber.

Chapter 8

Looking Downstream

The StarGlobe system is designed for distributed, highly adaptive in-network query processing on XML streams in P2P networks. Regardless of its computational power, every client is able to register a subscription (a query) on streams in the network. A query optimizer positions the operators (user-defined functions, selections, projections, etc.) in the network in a cost-effective way (e. g., considering network traffic). With reference to the needs of the virtual observatories described in this thesis and the results achieved so far, it is worth putting effort in further interdisciplinary cooperations.

Using StarGlobe to process the different catalogs in a distributed and parallel fashion and to compute results within the network leads to several benefits for the astrophysicist:
Nowadays huge catalogs have to be copied via FTP, DVD or other media to each researcher individually which immediately leads to data duplication. By streaming the data into a subscription-capable network the duplication is removed on common paths in the network. By providing mobile operators (user-defined functions), StarGlobe can be extended to process streams for special purposes.

Distributed data processing, manually-optimized query execution plans with dynamically loaded user-defined operations, and a graphical user interface to monitor and evaluate the infrastructure are the contributions by this thesis to the StarGlobe system.

Data stream management systems provide several benefits in various research communities as we described with two exemplary applications in the field of astrophysics.

This contention is supported by the results of our small prototypical scenario and the feedback provided by our cooperation partners and attendees at our demonstration at the VLDB 2005.

Support for multiple input streams and an implementation of fuzzy joins is certainly one of the next challenges we need to solve. Then current applications of the scientist can be compared with our infrastructure. When users install their plans, details must be provided by themselves. Whether some parts of the

plans such as the DTDs for operators can be derived automatically during plan installation or by a preprocessor is worth being considered in future work. This would alleviate users and would lower the barrier to use our infrastructure.

During the development phase mainly persistent data like files were used to simulate data streams. How modern features of database systems can be exploited for astrophysical applications and data residing in databases is extracted as XML stream in a generic fashion would complete the integration of data streams and persistent data and well deserves separate treatment.

When the ideas sketched in section 4.4.3 are realized and plans are also described in a declarative fashion, we can optimize query execution plans with user-defined operations. If research on what kinds of optimization, cost models, and statistics can be provided for user-defined execution plans heralds promising techniques, these will be integrated in StarGlobe.

Distributed processing of data streams in P2P networks will offer many benefits and will invite many research communities to participate in distributed collaboration.

Appendix A

Installing the Grid on Blades

The Globus-Toolkit 3.2.1 is installed on the blade server at the Chair for Informatics III - Database Systems of the Technische Universität München (TUM).

A.1 Installation Issues

When looking at the installation process of Globus Toolkit, there are significant differences between installing Globus on a blade server or on ordinary server farms or distributed computers.

Blades bring with them one or several CPU(s), a network interface card, and a local hard drive, yet share power supply, network, fans, and a *Network Attached Storage (NAS)*. The latter is storage shared among all blades. Assuming a service is running on a blade and this blade crashes, the service can boot on a different blade, and all request are re-routed to this new blade. The service however does not need to be copied when it resides on the NAS.

In the context of Grid Computing a blade server can play several roles. On the one hand, the owning institution can be part of a global federation and shares the blade with other cooperative institutions. On the other, the blade architecture might be used to simulate a federation of institutions.

The requirements for the administration are twofold. First, the administration should be as easy as possible. The workload when installing updates or new services should be kept at an minimum, yet provide a reasonable amount of flexibility. Second, as several people are working in the project in parallel and all should be able to use the blade server architecture for testing purposes, the installation process should not afford too much (if any) administrative rights on the blade server.

Taking all these requirements into account, one could either install the Globus Toolkit on each blade individually or install it on the NAS. The Stream- and StarGlobe systems use the Web Service architecture (the Grid Services) that was introduced with the Globus Toolkit 3 (GT3). The GT3 can be installed including

the services before Web Services were integrated (pre-WS services) or or only the Web Service Core (WS Core) which is implemented in Java.

These options and the consequences resulting from a specific choice are discussed in more detail in the following section.

A.1.1 Installation location

Before we focus on the individual Globus installation options let us concentrate on the choice where the Toolkit should be installed.

The first option would be to install the Globus Toolkit on every blade individually. This means, the administrator installs the GT3 on the local hard drive of the blade. This installation process resembles the approach when traditional servers are used and does not take much advantage of the blade architecture.

It certainly offers the greatest flexibility. Each blade can be configured individually. Unfortunately, the price for this flexibility is very high with respect to administration. Updates must be installed on every single blade. What happens when you installed a certain service only on a single blade and its local hard drive crashes? The service must be installed on a different blade.

To install the Globus Toolkit on the NAS virtually leads to "install once, run everywhere". As every blade accesses the same installation, updates can be done for all blades simultaneously.

This comes in handy for the fundamental services but might be surplus effort when services are needed only once in a network.

A.1.2 Package Options

Globus Toolkit offers several distributions. We are interested in the full distribution and the core distribution and describe which suits best the requirements of the different scenarios.

The Globus-Toolkit-3-Full-Installer is a large installation and contains a lot of services which are important, if the services are used in a productive environment but are of less importance for the development of the Grid Services themselves.

One of these components only available in the full-installer is a Certificate Authority (CA) that generates all the host-certificates and user-certificates used in the security layer of OGSA. This imposes quite a workload on the individual developer when planning to run StarGlobe on several blade servers.

The pre-WS services have to be compiled (takes a few hours), the CA and the individual host certificates must be installed.

The *core distribution* contains the basic APIs of OGSA implemented in Java and is denoted as Web Service Core (WS Core).

The installation process of the WS Core consists of extracting the archive, running the installation script and setting up environmental variables.

The current StarGlobe implementation only makes use of these basic APIs. Therefore developers can install the core distribution if only the StarGlobe system is under test and not necessarily the full distribution.

A.1.3 CA Installation

During the trail of the options from above issues originating from host-specific components as security and logging mechanisms were solved.

The security infrastructure is provided by a subcomponent below the GT3-Core and thus is necessary to be installed separately.

In some C header files there are references to the /etc/grid-security/ folder which should contain the mapping from users to certificates and the host-certificate of the machine. In general you can specify parameters to retrieve the credentials from different locations yet some components have wired the path to this directory and it seems quite an effort to replace all occurrences in the configuration files.

A.1.4 Directory Structure

The *full* Globus Toolkit 3 is installed on the blade system under the user account *globus*. Its home directory (/home/globus) has the following structure:

instance contains the globus installation.

packages contains the archives and scripts for installing Globus, CA, host-certificates, ...

packages/install-globus.sh script for installing globus.

install-ca.sh script for installing the CA, which is necessary only once per installation.

packages/install-hostcert.sh script for installing host-certificates which is necessary each time a new host is added to the Grid.

To install the Java-WS Core Globus Toolkit 3, decompress the downloaded archive, set the GLOBUS_LOCATION environment variable on the directory where the archive was decompressed to. Then run ant setup to generate the platform-specific scripts.

To start the globus instance, bin/globus-start-container.sh needs to be executed from GLOBUS_LOCATION.

A.2 Changing the Globus - Migration Issues to Globus Toolkit 4

In April 2005 the Globus Alliance has published the Globus Toolkit 4 (GT4). What would be the impact on the StarGlobe and the StreamGlobe system respectively, when we would switch to Globus Toolkit 4? We give a short summary according to the Migration Guide from the Globus web site.[1]

With the new version, the Globus Toolkit uses the *Web Service Resource Framework (WSRF)* and thus no longer specifies its services in the "home grown" GWSDL but in standard WSDL.

The change in PortTypes is also quite significant. The GridService port type implemented some basic OGSI functionality and so every Grid service either was a subclass of GridService or delegated this method calls accordingly. Another central feature in the OGSA framework was the ability to specify service factories to create several service instances on demand during runtime. Our Peer GridService is implemented in that fashion. Unfortunately, from Globus Toolkit 4 this is no longer supported.

Whereas in GT3 business logic and state handling of a GridService was highly coupled together into one class, these two concepts are decoupled in GT4. Business logic is written in a stateless *service class* and all state handling is implemented in a stateful *resource class*.

The separation of state and logic also results in the extraction of the state-handling from the deployment descriptor server-config.wsdd into a second configuration file (jndi-config.xml).

Whether these changes are significant to StreamGlobe and StarGlobe respectively remains to be seen.

[1] http://www.globus.org/toolkit/docs/4.0/migration_guide_gt3.html

Appendix B

Execution Plan XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- $Id: plan-schema.xsd,v 1.1.2.10 2005/09/15 13:22:56 hubers Exp
    $ -->
<xs:schema xmlns:pdc="urn:streamglobe.in.tum.de/pdc" xmlns:xs="http
    ://www.w3.org/2001/XMLSchema" targetNamespace="urn:streamglobe.in
    .tum.de/pdc" elementFormDefault="qualified">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      This schema defines the XML Schema for a plan distributed by the
          PlanDistributionComponent.
    </xs:documentation>
  </xs:annotation>
  <xs:element name="plan">
    <xs:annotation>
      <xs:documentation xml:lang="en">
    The root element of a query plan.
      </xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:annotation>
          <xs:documentation xml:lang="en">
      At each Peer operators can be added in the add-Element or
      deleted in the delete-Element. The sub-plans the operators are
          depending on
      are specified in the following plan elements.
        </xs:documentation>
        </xs:annotation>
        <xs:element name="add" type="pdc:addOperatorsAtPeerType"
            minOccurs="0"/>
        <xs:element name="delete" type="pdc:deleteOperatorsAtPeerType"
             minOccurs="0"/>
        <xs:element ref="pdc:plan" minOccurs="0" maxOccurs="unbounded
            "/>
      </xs:sequence>
      <xs:attribute name="atPeer" type="xs:string" use="required"/>
```

```
    <xs:attribute name="id" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:key name="planIdOrOperatorId">
    <xs:selector xpath="./pdc:add/pdc:streamoperator|./pdc:plan"/>
    <xs:field xpath="@id"/>
  </xs:key>
  <xs:keyref name="validReferences" refer="pdc:planIdOrOperatorId">
    <xs:selector xpath="./pdc:add/pdc:streamoperator/pdc:
       dependencies/pdc:streamreference"/>
    <xs:field xpath="@id"/>
  </xs:keyref>
  <xs:unique name="doNotDeleteNewOperators">
    <xs:selector xpath=".//pdc:streamoperator"/>
    <xs:field xpath="@id"/>
  </xs:unique>
</xs:element>
<xs:complexType name="addOperatorsAtPeerType">
  <xs:sequence>
    <xs:element name="streamoperator" type="pdc:
       abstractStreamoperatorType" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="deleteOperatorsAtPeerType">
  <xs:sequence>
    <xs:element name="streamoperator" maxOccurs="unbounded">
      <xs:complexType>
        <xs:attribute name="id" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="parameterType">
  <xs:sequence>
    <xs:element name="key" type="xs:string"/>
    <xs:element name="value" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="abstractStreamoperatorType" abstract="true">
  <xs:annotation>
    <xs:documentation xml:lang="en">
  generalization of a streamoperator
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="dependencies">
      <xs:annotation>
        <xs:documentation xml:lang="en">
    declares the streams this operator depends on.
      </xs:documentation>
      </xs:annotation>
      <xs:complexType>
```

```
      <xs:choice maxOccurs="unbounded">
        <xs:element name="streamreference" type="pdc:
            streamreferenceType">
          <xs:annotation>
            <xs:documentation xml:lang="en">
        define this element, if you reference a stream from a
        different operator or plan.
        when supported by StreamGlobe, multiple input streams are
        allowed with setting maxOccurs to a higher level as 1.
     </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="stream" type="pdc:streamType">
          <xs:annotation>
            <xs:documentation xml:lang="en">
        define this element, if you want to introduce a new
        stream into the plan.
     </xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:choice>
     </xs:complexType>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="streamreferenceType">
  <xs:attribute name="id" type="xs:string" use="required">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      this attribute defines which stream is referenced by this
          operator.
      You can reference either a different operator (for pipelining)
          ,
      or a different incoming stream.
  </xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>
<xs:complexType name="streamType">
  <xs:attribute name="id" type="xs:string" use="required">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      if you want to use a stream not yet defined in the plan,
      define the id in this attribute.
  </xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:complexType>
<xs:complexType name="queryStreamoperatorType">
  <xs:annotation>
```

```
    <xs:documentation xml:lang="en">
  definition for an XQuery-operator.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="pdc:abstractStreamoperatorType">
      <xs:sequence>
        <xs:element name="source" type="xs:string"/>
        <xs:element name="input-dtd" type="xs:string" minOccurs="0"/>
        <xs:element name="output-dtd" type="xs:string" minOccurs
           ="0"/>
        <xs:element name="userdefinedfunction" minOccurs="0"
           maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="name" type="xs:string" use="required
               "/>
            <xs:attribute name="codebase" type="pdc:codebaseType" use
               ="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" default="query"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="builtInStreamoperatorType">
  <xs:annotation>
    <xs:documentation xml:lang="en">
  builtIn operators. This element has to be extended
  whenever new built-in operators are created.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="pdc:abstractStreamoperatorType">
      <xs:attribute name="name" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:NMTOKEN">
            <xs:enumeration value="forward"/>
            <xs:enumeration value="display"/>
            <xs:enumeration value="statistic"/>
            <xs:enumeration value="counter"/>
            <xs:enumeration value="null"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="hopStreamoperatorType">
  <xs:annotation>
    <xs:documentation xml:lang="en">
```

```
    Used as preprocessor for removing redundant aggregation results
    before reusing. It selects bulding blocks in certain distance,
    like hopping, and forwards these blocks, so the name.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="pdc:abstractStreamoperatorType">
      <xs:sequence>
        <xs:element name="blocktag" type="xs:string"/>
        <xs:element name="step" type="xs:int"/>
      </xs:sequence>
      <xs:attribute name="name" default="hop"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="externalStreamoperatorType">
  <xs:complexContent>
    <xs:extension base="pdc:abstractStreamoperatorType">
      <xs:sequence>
        <xs:annotation>
          <xs:documentation xml:lang="en">
            At the moment inputStreamData is mandatory. When the
               handling
            class is more sophisticated and retrieves the dtd for the
               input
            streams from e.g., the SpeakerPeer and thus leaverages the
                user
            from tedious work it will be optional.
            When the system can build the output dtd on its own, the
            "outputstreamdata" element can be optional, too.
          </xs:documentation>
        </xs:annotation>
        <xs:element name="authorizedby" type="xs:string">
          <xs:annotation>
            <xs:documentation xml:lang="en">
              The authorizedby-attribute should adhere to the
              DistinguishedName patterns as described in RFC2253.
            </xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="inputstreamdata" type="pdc:
           inputStreamDataType" maxOccurs="unbounded"/>
        <xs:element name="outputstreamdata" type="pdc:
           outputStreamDataType"/>
        <xs:element name="parameter" type="pdc:parameterType"
           minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="codebase" type="pdc:codebaseType" use="
         required"/>
      <xs:attribute name="dependencyToEnrich" type="xs:string" use="
```

```
        optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="streamDataType">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      Information specified for an external operator.
      When an empty dtd is specified the StarGlobe-System
      tries to resolve it.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="dtd" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="inputStreamDataType">
  <xs:annotation>
    <xs:documentation xml:lang="en">
      each instance has to reference a certain stream
      and can optional specify typed mappings to retrieve
      data and pass it to the specified StreamIterator.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="pdc:streamDataType">
      <xs:sequence>
        <xs:element name="variable" type="pdc:typedVariableMapping"
           minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="id" type="xs:string" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="outputStreamDataType">
  <xs:complexContent>
    <xs:extension base="pdc:streamDataType">
      <xs:sequence>
        <xs:element name="variable" type="pdc:untypedVariableMapping"
            minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:complexType name="untypedVariableMapping">
  <xs:attribute name="name" use="required"/>
  <xs:attribute name="select" type="pdc:mappingPathType" use="
     required"/>
  <xs:attribute name="position" use="optional">
    <xs:simpleType>
      <xs:restriction base="xs:NMTOKEN">
```

```
        <xs:enumeration value="FIRST"/>
        <xs:enumeration value="LAST"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
 </xs:complexType>
 <xs:complexType name="typedVariableMapping">
  <xs:complexContent>
   <xs:extension base="pdc:untypedVariableMapping">
    <xs:attribute name="type" use="required"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:simpleType name="mappingPathType">
  <xs:annotation>
   <xs:documentation xml:lang="en">
    only mappingpaths starting with a "./" are supported.
   </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
   <xs:pattern value="\./.*"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="codebaseType">
  <xs:annotation>
   <xs:documentation xml:lang="en">
    Only URIs using the file- or the http-protocol are supported.
   </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:anyURI">
   <xs:pattern value="http://.*"/>
   <xs:pattern value="file:///.*"/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

Appendix C

A-Star Workflow

The scenario (*plan1scenario.xml*) and the query execution plan (*astarworkflow.xml*) for the A-Star workflow are located in *docs/usecases/scenario/* of the *plandistribution* branch of the *streamglobe/grid-service* module in the Streamglobe CVS repository.

C.1 Scenario

```
<?xml version="1.0" ?>
<scenario name="sample" xmlns="http://streamglobe.net/scenario">
   <!-- Resources defining the input streams -->
   <graph>
      <vertex vid="0" label="Peer0"/>
      <vertex vid="1" label="Peer1"/>
      <vertex vid="2" label="Peer2"/>
      <vertex vid="3" label="Peer3"/>
      <edge source="0" target="1"/>
      <edge source="1" target="2"/>
      <edge source="2" target="3"/>
   </graph>
   <streams>
      <kindDefinition>
         <kind name="file" class="streamglobe.client.p2p.
            FileContentServer"/>
      </kindDefinition>
      <stream sid="Stream-0" type="file">
         <dtd filename="@DTD_PATH@"/>
         <param name="stream.filename">
            @FILE_PATH@
         </param>
         <param name="stream.server.port">9009</param>
         <param name="stream.sleep.time">2000</param>
      </stream>
   </streams>
   <injectorMapping>
```

```
      <mapping peer="0" stream="Stream-0"/>
   </injectorMapping>
</scenario>
```

C.2 XML Execution Plan

```
<?xml version="1.0" encoding="UTF-8"?>
<plan atPeer="@Peer3-GSH@" id="stream-4" xmlns="urn:streamglobe.in.
    tum.de/pdc"
  xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
   <streamoperator id="result-display"
     xsi:type="builtInStreamoperatorType" name="display">
     <dependencies>
       <streamreference id="stream-3"/>
     </dependencies>
   </streamoperator>
  </add>
      <plan atPeer="@Peer2-GSH@" id="stream-3"
        xmlns="urn:streamglobe.in.tum.de/pdc"
        xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <add>
           <streamoperator id="mahalanobis" codebase="@GAVO_JAR@"
             name="org.gavo.streamoperators.
                SimpleMahalanobisDistanceIterator"
             xsi:type="externalStreamoperatorType">
             <dependencies>
               <stream id="stream-2"/>
             </dependencies>
             <authorizedby>Tobias Scholl</authorizedby>
             <inputstreamdata id="stream-2">
               <dtd>
               <![CDATA[
               <!ELEMENT photons (photon)*>
               <!ELEMENT photon (coord, phc, en, det_time, cartesian)>
               <!ELEMENT coord (cel, det)>
               <!ELEMENT cel (ra, dec)>
               <!ELEMENT ra (#PCDATA)>
               <!ELEMENT dec (#PCDATA)>
               <!ELEMENT det (dx, dy)>
               <!ELEMENT dx (#PCDATA)>
               <!ELEMENT dy (#PCDATA)>
               <!ELEMENT phc (#PCDATA)>
               <!ELEMENT en (#PCDATA)>
               <!ELEMENT det_time (#PCDATA)>
               <!ELEMENT cartesian (x, y, z)>
               <!ELEMENT x (#PCDATA)>
               <!ELEMENT y (#PCDATA)>
               <!ELEMENT z (#PCDATA)>
```

```
        ]]>
        </dtd>
        <variable name="POINT[0]" select="./cartesian/x" type="
           Double"/>
        <variable name="POINT[1]" select="./cartesian/y" type="
           Double"/>
        <variable name="POINT[2]" select="./cartesian/z" type="
           Double"/>
        <variable name="SPHERIC[0]" select="./coord/cel/ra"
           type="Double"/>
        <variable name="SPHERIC[1]" select="./coord/cel/dec"
           type="Double"/>
      </inputstreamdata>
      <outputstreamdata>
        <dtd/>
        <variable name="DISTANCE" select="./dist/mahalanobis"/>
        <variable name="POS_ANGLE" select="./dist/pd"/>
      </outputstreamdata>
      <parameter>
        <key>REF_X</key>
        <value>0.5589609245067093</value>
      </parameter>
      <parameter>
        <key>REF_Y</key>
        <value>-0.619582747878259</value>
      </parameter>
      <parameter>
        <key>REF_Z</key>
        <value>0.5510715955356713</value>
      </parameter>
      <parameter>
        <key>SIGMA</key>
        <value>0.05</value>
      </parameter>
    </streamoperator>
  </add>
  <plan atPeer="@Peer1-GSH@" id="stream-2"
  xmlns="urn:streamglobe.in.tum.de/pdc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <add>
    <streamoperator id="cone" codebase="@GAVO_JAR@"
      name="org.gavo.streamoperators.
         SimpleConeSearchStreamIterator"
      xsi:type="externalStreamoperatorType">
      <dependencies>
        <streamreference id="stream-1"/>
      </dependencies>
      <authorizedby>Tobias Scholl</authorizedby>
      <inputstreamdata id="stream-1">
        <dtd>
        <![CDATA[
```

```
      <!ELEMENT photons (photon)*>
      <!ELEMENT photon (coord, phc, en, det_time, cartesian)>
      <!ELEMENT coord (cel, det)>
      <!ELEMENT cel (ra, dec)>
      <!ELEMENT ra (#PCDATA)>
      <!ELEMENT dec (#PCDATA)>
      <!ELEMENT det (dx, dy)>
      <!ELEMENT dx (#PCDATA)>
      <!ELEMENT dy (#PCDATA)>
      <!ELEMENT phc (#PCDATA)>
      <!ELEMENT en (#PCDATA)>
      <!ELEMENT det_time (#PCDATA)>
      <!ELEMENT cartesian (x, y, z)>
      <!ELEMENT x (#PCDATA)>
      <!ELEMENT y (#PCDATA)>
      <!ELEMENT z (#PCDATA)>
      ]]>
      </dtd>
      <variable name="POINT_X" select="./cartesian/x" type="
          Double"/>
      <variable name="POINT_Y" select="./cartesian/y" type="
          Double"/>
      <variable name="POINT_Z" select="./cartesian/z" type="
          Double"/>
    </inputstreamdata>
    <outputstreamdata>
      <dtd/>
    </outputstreamdata>
    <parameter>
      <key>CENTER_X</key>
      <value>0.5589609245067093</value>
    </parameter>
    <parameter>
      <key>CENTER_Y</key>
      <value>-0.619582747878259</value>
    </parameter>
    <parameter>
      <key>CENTER_Z</key>
      <value>0.5510715955356713</value>
    </parameter>
    <parameter>
      <key>SEARCH_RADIUS</key>
      <value>0.5</value>
    </parameter>
  </streamoperator>
</add>
<plan id="stream-1" atPeer="@Peer0-GSH@"
xmlns="urn:streamglobe.in.tum.de/pdc"
  xmlns:pdc="urn:streamglobe.in.tum.de/pdc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <add>
```

```
            <streamoperator id="transform" codebase="@GAVO_JAR@"
              name="org.gavo.streamoperators.CoordinateStreamIterator
                 "
              xsi:type="externalStreamoperatorType">
              <dependencies>
                <stream id="stream-0"/>
              </dependencies>
              <authorizedby>Tobias Scholl</authorizedby>
              <inputstreamdata id="stream-0">
                <dtd>
                <![CDATA[
                <!ELEMENT photons (photon)*>
                <!ELEMENT photon (coord, phc, en, det_time)>
                <!ELEMENT coord (cel, det)>
                <!ELEMENT cel (ra, dec)>
                <!ELEMENT ra (#PCDATA)>
                <!ELEMENT dec (#PCDATA)>
                <!ELEMENT det (dx, dy)>
                <!ELEMENT dx (#PCDATA)>
                <!ELEMENT dy (#PCDATA)>
                <!ELEMENT phc (#PCDATA)>
                <!ELEMENT en (#PCDATA)>
                <!ELEMENT det_time (#PCDATA)>
                ]]>
                </dtd>
                <variable name="RA" select="./coord/cel/ra" type="
                   Double"/>
                <variable name="DEC" select="./coord/cel/dec" type="
                   Double"/>
              </inputstreamdata>
              <outputstreamdata>
                <dtd/>
                <variable name="cartesianCoordinates[0]" select="./
                   cartesian/x"/>
                <variable name="cartesianCoordinates[1]" select="./
                   cartesian/y"/>
                <variable name="cartesianCoordinates[2]" select="./
                   cartesian/z"/>
              </outputstreamdata>
            </streamoperator>
          </add>
        </plan>
      </plan>
    </plan>
</plan>
```

C.3 User-Defined Operators

C.3.1 Coordinate Transformation

```
package org.gavo.streamoperators;
import org.apache.commons.beanutils.DynaBean;
import org.gavo.util.math.CoordinateTransformation;

public class CoordinateStreamIterator implements StreamIterator {
    public static final String RA = "RA";
    public static final String DEC = "DEC";
    public static final String CARTESIAN = "cartesianCoordinates";
    private StreamWriter writer;

    public void open(DynaBean config, StreamWriter writer) {
        this.writer = writer;
    }

    public synchronized void next(StreamIteratorEvent nextItem) {
        DynaBean params = nextItem.getParameters();
        Double ra = (Double) params.get(RA);
        Double dec = (Double) params.get(DEC);

        double[] cartesian = CoordinateTransformation.toCartesian(
                    ra.doubleValue(), dec.doubleValue());

        for (int i = 0; i < cartesian.length; i++) {
            params.set(CARTESIAN, i, String.valueOf(cartesian[i]));
        }
        writer.write(nextItem);
    }

    public synchronized void close(String StreamId) {
    }
}
```

C.3.2 Simple Cone Search

```
package org.gavo.streamoperators;

import org.apache.commons.beanutils.DynaBean;
import org.gavo.util.math.SphericalTrigonometry;
import org.gavo.util.math.Trigonometry;

import streamglobe.services.p2p.engine.operator.starglobe.StreamIterator;
import streamglobe.services.p2p.engine.operator.starglobe.
    StreamIteratorEvent;
```

```
import streamglobe.services.p2p.engine.operator.starglobe.StreamWriter;

/**
 * Calculates a simple cone search around a point with given radius.
 * Parameters are expected to be in cartesian coordinates, the radius in arc.
 *
 * For this purpose the arcDistance between the Cone-Center and a point from
     the
 * stream is calculated. If it is greater than the the maximal distance, it will
 * be returned to the output stream. Otherwise not.
 */
public class SimpleConeSearchStreamIterator implements StreamIterator {

    // input parameters
    public static final String POINT_X = "POINT_X";
    public static final String POINT_Y = "POINT_Y";
    public static final String POINT_Z = "POINT_Z";

    // config parameters
    public static final String CENTER_X = "CENTER_X";
    public static final String CENTER_Y = "CENTER_Y";
    public static final String CENTER_Z = "CENTER_Z";
    public static final String SEARCH_RADIUS = "SEARCH_RADIUS";

    private double[] center = new double[3];
    private double maxDistance;

    private StreamWriter writer;

    public void open(DynaBean config, StreamWriter writer) {
        center[0] = Double.parseDouble((String) config.get(CENTER_X));
        center[1] = Double.parseDouble((String) config.get(CENTER_Y));
        center[2] = Double.parseDouble((String) config.get(CENTER_Z));
        Double radius = Double.valueOf((String) config.get(SEARCH_RADIUS
            ));

        maxDistance = Math.cos(radius.doubleValue());
        this.writer = writer;
    }

    public void next(StreamIteratorEvent nextItem) {
        DynaBean parameters = nextItem.getParameters();
        Double pointX = (Double) parameters.get(POINT_X);
```

```
        Double pointY = (Double) parameters.get(POINT_Y);
        Double pointZ = (Double) parameters.get(POINT_Z);

        double[] point = new double[3];
        point[0] = pointX.doubleValue();
        point[1] = pointY.doubleValue();
        point[2] = pointZ.doubleValue();

        double angle = SphericalTrigonometry.arcDistance(center, point);

        if (Trigonometry.cos(angle) > maxDistance) {
            writer.write(nextItem);
        }
    }

    public void close(String streamId) {
    }
}
```

C.3.3 Mahalanobis Distance

```
package org.gavo.streamoperators;

import java.util.ArrayList;
import java.util.List;

import org.apache.commons.beanutils.DynaBean;
import org.gavo.util.math.CoordinateTransformation;
import org.gavo.util.math.SphericalTrigonometry;

import streamglobe.services.p2p.engine.operator.starglobe.StreamIterator;
import streamglobe.services.p2p.engine.operator.starglobe.
    StreamIteratorEvent;
import streamglobe.services.p2p.engine.operator.starglobe.StreamWriter;

/**
 * Calculates the MahalanobisDistance and the polar angle between a point
     given
 * at initialisation time and one point from the stream.
 *
 * It is assumed that only one sigma for a stream is specified. If none is
 * specified the Eukledian distance is calculated.
 *
```

```
 */
public class SimpleMahalanobisDistanceIterator implements StreamIterator
    {

    // config parameters
    public static final String REF_X = "REF_X";
    public static final String REF_Y = "REF_Y";
    public static final String REF_Z = "REF_Z";
    public static final String SIGMA = "SIGMA";

    // input parameters

    // cartesian coordinates
    public static final String POINT = "POINT";

    // spherical coordinates
    public static final String SPHERIC = "SPHERIC";

    // output parameters
    public static final String DISTANCE = "DISTANCE";
    public static final String POS_ANGLE = "POS_ANGLE";

    private double[] refPointCartesian;
    private double[] refPointSpheric;
    private double sigma = 1.0;
    private StreamWriter writer;

    public void open(DynaBean config, StreamWriter writer) {
        List referencePoint = new ArrayList();
        referencePoint.add(Double.valueOf((String) config.get(REF_X)));
        referencePoint.add(Double.valueOf((String) config.get(REF_Y)));
        referencePoint.add(Double.valueOf((String) config.get(REF_Z)));

        refPointCartesian = getDoubleArray(referencePoint);
        refPointSpheric = CoordinateTransformation.toSpherical(
            refPointCartesian);

        if (config.get(SIGMA) != null) {
            sigma = Double.parseDouble((String) config.get(SIGMA));
        }
        this.writer = writer;
    }
```

```
    /**
     * returns a double array from the List containing Double objects.
     * @param doubleList
     */
    private double[] getDoubleArray(List doubleList) {
        double[] result = new double[doubleList.size()];
        for (int i = 0; i < doubleList.size(); i++) {
            result[i] = ((Double) doubleList.get(i)).doubleValue();
        }
        return result;
    }

    public void next(StreamIteratorEvent nextItem) {
        DynaBean parameters = nextItem.getParameters();

        double[] cartesian;
        double[] spherical;
        List pointCartesian = (List) parameters.get(POINT);
        cartesian = getDoubleArray(pointCartesian);

        List sphericalCoords = (List) parameters.get(SPHERIC);
        spherical = getDoubleArray(sphericalCoords);

        double arcDistance = SphericalTrigonometry.arcDistance(
            refPointCartesian, cartesian);
        double mahalanobis = arcDistance / sigma;
        double polarAngle = SphericalTrigonometry.positionAngle(
            refPointSpheric, spherical);

        parameters.set(DISTANCE, String.valueOf(mahalanobis));
        parameters.set(POS_ANGLE, String.valueOf(polarAngle));

        writer.write(nextItem);
    }

    public void close(String StreamId) {
    }
}
```

Appendix D

VLDB 2005 Scenarios

The following scenarios are provided in the *etc/vldb2005/templates* folder of the *streamglobe/gui* module in the StreamGlobe CVS repository.

D.1 Vela Scenario

Filename poster-scenario.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="010" />
  <vertex vid="1" label="100" />
  <vertex vid="2" label="110" />
  <vertex vid="3" label="001" />
  <vertex vid="4" label="011" />
  <vertex vid="5" label="111" />
  <vertex vid="6" label="101" />
  <vertex vid="7" label="000" />
  <edge source="4" target="0" />
  <edge source="5" target="2" />
  <edge source="6" target="3" />
  <edge source="1" target="7" />
  <edge source="0" target="7" />
  <edge source="5" target="4" />
  <edge source="2" target="0" />
  <edge source="2" target="1" />
  <edge source="4" target="3" />
  <edge source="5" target="6" />
  <edge source="6" target="1" />
```

```
  <edge source="3" target="7" />
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="@CONTENT_SERVER_CLASS@" />
  </kindDefinition>
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela_demo.xml</
       param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">100</param>
  </stream>
</streams>

<queries>
  <query qid="1">
    <![CDATA[
<photons>
{
  for $p in stream("stream-0")/photons/photon
  where $p/coord/cel/ra >= 120.0
    and $p/coord/cel/ra <= 138.0
    and $p/coord/cel/dec >= -49.0
    and $p/coord/cel/dec <= -40.0
  return
    <vela_photon>
      {$p/coord/cel/ra} {$p/coord/cel/dec}
      {$p/phc} {$p/en} {$p/det-time}
    </vela_photon>
}
</photons>
    ]]>
  </query>

  <query qid="2">
    <![CDATA[
<photons>
{
  for $p in stream("stream-0")/photons/photon
  where $p/en >= 1.3
    and $p/coord/cel/ra >= 130.5
    and $p/coord/cel/ra <= 135.5
    and $p/coord/cel/dec >= -48.0
    and $p/coord/cel/dec <= -45.0
  return
    <rxj_photon>
      {$p/coord/cel/ra} {$p/coord/cel/dec}
      {$p/en} {$p/det-time}
    </rxj_photon>
```

```
}
</photons>
   ]]>
  </query>

  <query qid="3">
   <![CDATA[
<photons>
{
  for $w in stream("stream-0")/photons/photon
   [en >= 1.3
   and coord/cel/ra >= 130.5
   and coord/cel/ra <= 135.5
   and coord/cel/dec >= -48.0
   and coord/cel/dec <= -45.0]
   | count 20 step 10 |
  let $a := avg($w/photon/en)
  return
   <avg_en>
    {$a}
   </avg_en>
}
</photons>
   ]]>
  </query>

  <query qid="4">
   <![CDATA[
<photons>
{
  for $w in stream("stream-0")/photons/photon
   [en >= 1.3
   and coord/cel/ra >= 130.5
   and coord/cel/ra <= 135.5
   and coord/cel/dec >= -48.0
   and coord/cel/dec <= -45.0]
   | count 60 step 40 |
  let $a := avg($w/photon/en)
  return
   <avg_en>
    {$a}
   </avg_en>
}
</photons>
   ]]>
  </query>

</queries>

<injectorMapping>
  <mapping peer="4" stream="stream-0" />
```

```
</injectorMapping>

<queryMapping>
  <mapping peer="0" query="1" />
  <mapping peer="2" query="2" />
  <mapping peer="1" query="3" />
  <mapping peer="6" query="4" />
</queryMapping>

</scenario>
```

D.2 Throughput Scenario

Filename throughput.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="010" />
  <vertex vid="1" label="100" />
  <vertex vid="2" label="110" />
  <vertex vid="3" label="001" />
  <vertex vid="4" label="011" />
  <vertex vid="5" label="111" />
  <vertex vid="6" label="101" />
  <vertex vid="7" label="000" />
  <edge source="4" target="0" />
  <edge source="5" target="2" />
  <edge source="6" target="3" />
  <edge source="1" target="7" />
  <edge source="0" target="7" />
  <edge source="5" target="4" />
  <edge source="2" target="0" />
  <edge source="2" target="1" />
  <edge source="4" target="3" />
  <edge source="5" target="6" />
  <edge source="6" target="1" />
  <edge source="3" target="7" />
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="@CONTENT_SERVER_CLASS@" />
  </kindDefinition>
```

```
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela_demo.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">10</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
</streams>

<queries>
  <query qid="0">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="1">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 260
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="2">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="3">
```

```
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="4">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="5">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="6">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="7">
```

```
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en} {$p/det-time}
  </vela>
   ]]>
  </query>

  <query qid="8">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en} {$p/det-time}
  </vela>
   ]]>
  </query>

  <query qid="9">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en}
  </vela>
   ]]>
  </query>

  <query qid="10">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en} {$p/det-time}
  </vela>
   ]]>
  </query>

  <query qid="11">
```

```
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="12">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="13">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 4 step 2|)
return <avg_en>{$p}</avg_en>
   ]]>
  </query>

  <query qid="14">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
```

```
  </query>

  <query qid="15">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 4 step 2|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="16">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="17">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="18">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
```

```
  </query>

  <query qid="19">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="20">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 260
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="21">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="22">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 260
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
```

```
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="23">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="24">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 260
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

</queries>

<injectorMapping>
  <mapping peer="0" stream="stream-0" />
</injectorMapping>

<queryMapping>
  <mapping peer="0" query="0" install_interval="0" />
  <mapping peer="1" query="1" install_interval="0" />
  <mapping peer="4" query="2" install_interval="0" />
  <mapping peer="2" query="3" install_interval="0" />
  <mapping peer="0" query="4" install_interval="0" />
  <mapping peer="7" query="5" install_interval="0" />
  <mapping peer="4" query="6" install_interval="0" />
  <mapping peer="4" query="7" install_interval="0" />
  <mapping peer="1" query="8" install_interval="0" />
```

```
  <mapping peer="5" query="9" install_interval="0" />
  <mapping peer="0" query="10" install_interval="0" />
  <mapping peer="4" query="11" install_interval="0" />
  <mapping peer="3" query="12" install_interval="0" />
  <mapping peer="5" query="13" install_interval="0" />
  <mapping peer="4" query="14" install_interval="0" />
  <mapping peer="2" query="15" install_interval="0" />
  <mapping peer="7" query="16" install_interval="0" />
  <mapping peer="1" query="17" install_interval="0" />
  <mapping peer="1" query="18" install_interval="0" />
  <mapping peer="6" query="19" install_interval="0" />
  <mapping peer="6" query="20" install_interval="0" />
  <mapping peer="5" query="21" install_interval="0" />
  <mapping peer="1" query="22" install_interval="0" />
  <mapping peer="6" query="23" install_interval="0" />
  <mapping peer="1" query="24" install_interval="0" />
</queryMapping>

</scenario>
```

D.3 Grid Scenario

Filename grid.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="(1, 0)" />
  <vertex vid="1" label="(0, 0)" />
  <vertex vid="2" label="(0, 2)" />
  <vertex vid="3" label="(2, 0)" />
  <vertex vid="4" label="(1, 1)" />
  <vertex vid="5" label="(2, 2)" />
  <vertex vid="6" label="(1, 2)" />
  <vertex vid="7" label="(0, 1)" />
  <vertex vid="8" label="(2, 1)" />
  <edge source="7" target="1" />
  <edge source="3" target="0" />
  <edge source="4" target="0" />
  <edge source="0" target="1" />
  <edge source="5" target="6" />
  <edge source="8" target="3" />
  <edge source="6" target="4" />
  <edge source="6" target="2" />
```

```
  <edge source="2" target="7" />
  <edge source="5" target="8" />
  <edge source="8" target="4" />
  <edge source="4" target="7" />
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="@CONTENT_SERVER_CLASS@" />
  </kindDefinition>
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela_demo.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">10</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
</streams>

<queries>
  <query qid="0">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="1">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 260
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="2">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
```

```
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
   ]]>
  </query>

  <query qid="3">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en} {$p/det-time}
  </vela>
   ]]>
  </query>

  <query qid="4">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en}
  </vela>
   ]]>
  </query>

  <query qid="5">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en}
  </vela>
   ]]>
  </query>

  <query qid="6">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
```

```
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="7">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="8">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="9">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="10">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
```

```
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="11">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="12">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="13">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 4 step 2|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="14">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
```

```
where $p/coord/cel/ra >= 0
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="15">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 4 step 2|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="16">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="17">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>
```

```
  <query qid="18">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="19">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="20">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 260
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="21">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
```

```
   ]]>
  </query>

  <query qid="22">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 260
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en} {$p/det-time}
  </vela>
   ]]>
  </query>

  <query qid="23">
   <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
   ]]>
  </query>

  <query qid="24">
   <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 30
  and $p/coord/cel/ra <= 260
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
   {$p/coord/cel/ra} {$p/coord/cel/dec}
       {$p/en} {$p/det-time}
  </vela>
   ]]>
  </query>

</queries>

<injectorMapping>
  <mapping peer="0" stream="stream-0" />
</injectorMapping>
```

```
<queryMapping>
  <mapping peer="0" query="0" install_interval="0" />
  <mapping peer="1" query="1" install_interval="0" />
  <mapping peer="4" query="2" install_interval="0" />
  <mapping peer="2" query="3" install_interval="0" />
  <mapping peer="0" query="4" install_interval="0" />
  <mapping peer="7" query="5" install_interval="0" />
  <mapping peer="4" query="6" install_interval="0" />
  <mapping peer="4" query="7" install_interval="0" />
  <mapping peer="1" query="8" install_interval="0" />
  <mapping peer="5" query="9" install_interval="0" />
  <mapping peer="0" query="10" install_interval="0" />
  <mapping peer="4" query="11" install_interval="0" />
  <mapping peer="3" query="12" install_interval="0" />
  <mapping peer="5" query="13" install_interval="0" />
  <mapping peer="4" query="14" install_interval="0" />
  <mapping peer="2" query="15" install_interval="0" />
  <mapping peer="7" query="16" install_interval="0" />
  <mapping peer="1" query="17" install_interval="0" />
  <mapping peer="1" query="18" install_interval="0" />
  <mapping peer="6" query="19" install_interval="0" />
  <mapping peer="6" query="20" install_interval="0" />
  <mapping peer="5" query="21" install_interval="0" />
  <mapping peer="1" query="22" install_interval="0" />
  <mapping peer="6" query="23" install_interval="0" />
  <mapping peer="1" query="24" install_interval="0" />
</queryMapping>

</scenario>
```

D.4 Linear Scenario

Filename linear.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="(1)" />
  <vertex vid="1" label="(3)" />
  <vertex vid="2" label="(2)" />
  <vertex vid="3" label="(0)" />
  <edge source="0" target="3" />
  <edge source="1" target="2" />
  <edge source="2" target="0" />
```

```
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="@CONTENT_SERVER_CLASS@" />
  </kindDefinition>
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela_demo.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">10</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
  <stream sid="stream-1" type="file">
    <dtd filename="etc/schemas/FIRST.dtd" />
    <param name="stream.filename">@FILE_DIR@/first/first-10MB.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_1@</param>
    <param name="stream.sleep.time">100</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
</streams>

<queries>
  <query qid="0">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.8
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="1">
    <![CDATA[
let $s := avg(stream("stream-1")/first/source
[pos/coords/ra >= 1.0
  and pos/coords/ra <= 140.0
  and pos/coords/dec >= -15.0
  and pos/coords/dec <= 20.0]/rms
  |count 8 step 2|)
return <avg_rms>{$s}</avg_rms>
    ]]>
  </query>

  <query qid="2">
    <![CDATA[
```

```
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 260
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>

  <query qid="3">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="4">
    <![CDATA[
let $s := avg(stream("stream-1")/first/source
[pos/coords/ra >= 1.0
  and pos/coords/ra <= 140.0
  and pos/coords/dec >= -15.0
  and pos/coords/dec <= 20.0]/rms
  |count 16 step 2|)
return <avg_rms>{$s}</avg_rms>
    ]]>
  </query>
</queries>

<injectorMapping>
  <mapping peer="3" stream="stream-0" />
  <mapping peer="1" stream="stream-1" />
</injectorMapping>

<queryMapping>
  <mapping peer="3" query="0" install_interval="0" />
  <mapping peer="0" query="1" install_interval="0" />
  <mapping peer="2" query="2" install_interval="0" />
  <mapping peer="3" query="3" install_interval="0" />
  <mapping peer="2" query="4" install_interval="0" />
</queryMapping>
```

```
</scenario>
```

D.5 Random Scenario

Filename random.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="(1, 0, 0)" />
  <vertex vid="1" label="(1, 0, 1)" />
  <vertex vid="2" label="(0, 1, 1)" />
  <vertex vid="3" label="(0, 0, 1)" />
  <edge source="0" target="1" />
  <edge source="2" target="1" />
  <edge source="2" target="0" />
  <edge source="3" target="1" />
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="@CONTENT_SERVER_CLASS@" />
  </kindDefinition>
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela_demo.xml</
       param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">10</param>
    <param name="stream.injection.mode">automatic</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
</streams>

<queries>
  <query qid="1">
    <![CDATA[
  for $p in stream("stream-0")/photons/photon
  where $p/coord/cel/ra >= 120
    and $p/coord/cel/ra <= 138
    and $p/coord/cel/dec >= -49
    and $p/coord/cel/dec <= -40
  return
    <vela_photon>
```

```
    {$p/coord/cel/ra} {$p/coord/cel/dec}
    {$p/phc} {$p/en} {$p/det-time}
  </vela_photon>
  ]]>
</query>

<query qid="2">
  <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 1.3
  and $p/coord/cel/ra >= 130.5
  and $p/coord/cel/ra <= 135.5
  and $p/coord/cel/dec >= -48.0
  and $p/coord/cel/dec <= -45.0
return
  <rxj_photon>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
    {$p/en} {$p/det-time}
  </rxj_photon>
  ]]>
</query>

<query qid="3">
  <![CDATA[
let $a := avg(stream("stream-0")/photons/photon
  [en >= 1.3
  and coord/cel/ra >= 130.5
  and coord/cel/ra <= 135.5
  and coord/cel/dec >= -48.0
  and coord/cel/dec <= -45.0]/en
  | count 20 step 10 |)
return
  <avg_en>
    {$a}
  </avg_en>
  ]]>
</query>

<query qid="4">
  <![CDATA[
let $a := avg(stream("stream-0")/photons/photon
  [en >= 1.3
  and coord/cel/ra >= 130.5
  and coord/cel/ra <= 135.5
  and coord/cel/dec >= -48.0
  and coord/cel/dec <= -45.0]/en
  | count 60 step 40 |)
return
  <avg_en>
    {$a}
  </avg_en>
```

```
   ]]>
  </query>

</queries>

<injectorMapping>
  <mapping peer="0" stream="stream-0" />
</injectorMapping>

<queryMapping>
  <mapping peer="2" query="1" />
  <mapping peer="0" query="2" />
  <mapping peer="1" query="3" />
  <mapping peer="3" query="4" />
</queryMapping>

</scenario>
```

D.6 Full-Demo Scenario

Filename full-demo.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="(2, 2, 1)" />
  <vertex vid="1" label="(1, 0, 1)" />
  <vertex vid="2" label="(3, 2, 1)" />
  <vertex vid="3" label="(2, 3, 1)" />
  <vertex vid="4" label="(0, 1, 1)" />
  <vertex vid="5" label="(0, 2, 1)" />
  <vertex vid="6" label="(2, 0, 1)" />
  <vertex vid="7" label="(1, 1, 1)" />
  <vertex vid="8" label="(3, 0, 1)" />
  <vertex vid="9" label="(3, 3, 1)" />
  <vertex vid="10" label="(2, 1, 1)" />
  <vertex vid="11" label="(1, 2, 1)" />
  <vertex vid="12" label="(0, 3, 1)" />
  <vertex vid="13" label="(0, 0, 1)" />
  <vertex vid="14" label="(3, 1, 1)" />
  <vertex vid="15" label="(1, 3, 1)" />
  <edge source="3" target="0" />
  <edge source="0" target="11" />
  <edge source="10" target="6" />
```

```
  <edge source="9" target="2" />
  <edge source="2" target="0" />
  <edge source="7" target="4" />
  <edge source="15" target="11" />
  <edge source="11" target="5" />
  <edge source="0" target="10" />
  <edge source="6" target="1" />
  <edge source="14" target="10" />
  <edge source="9" target="3" />
  <edge source="2" target="14" />
  <edge source="3" target="15" />
  <edge source="14" target="8" />
  <edge source="12" target="5" />
  <edge source="7" target="1" />
  <edge source="10" target="7" />
  <edge source="4" target="13" />
  <edge source="15" target="12" />
  <edge source="8" target="6" />
  <edge source="1" target="13" />
  <edge source="11" target="7" />
  <edge source="5" target="4" />
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="streamglobe.client.p2p.FileContentServer
        " />
  </kindDefinition>
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela-100MBa.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">10</param>
  </stream>
  <stream sid="stream-1" type="file">
    <dtd filename="etc/schemas/FIRST.dtd" />
    <param name="stream.filename">@FILE_DIR@/first/first-10MB.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_1@</param>
    <param name="stream.sleep.time">10</param>
  </stream>
  <stream sid="stream-2" type="file">
    <dtd filename="etc/schemas/sdss.dtd" />
    <param name="stream.filename">@FILE_DIR@/sdss/sdss-10MB.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_2@</param>
    <param name="stream.sleep.time">100</param>
  </stream>
</streams>
```

```
<queries>
  <query qid="0">
    <![CDATA[
for $s in stream("stream-2")/sdss/source
where $s/coord/ra >= 236
  and $s/coord/ra <= 240
  and $s/coord/dec >= -1.1
  and $s/coord/dec <= -0.8
return
<result>{$s/objID} {$s/coord/ra} {$s/coord/dec} {$s/type}</result>
    ]]>
  </query>

  <query qid="1">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 260
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>

  <query qid="2">
    <![CDATA[
let $p := avg(stream("stream-0")/photons/photon
[coord/cel/ra >= 30.0
  and coord/cel/ra <= 350.0
  and coord/cel/dec >= -78.0
  and coord/cel/dec <= 69.0]/en
  |count 2 step 1|)
return <avg_en>{$p}</avg_en>
    ]]>
  </query>

  <query qid="3">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/en >= 0.1
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en} {$p/det-time}
  </vela>
    ]]>
  </query>
```

```
  <query qid="4">
    <![CDATA[
let $s := avg(stream("stream-1")/first/source
[pos/coords/ra >= 0.0
  and pos/coords/ra <= 140.0
  and pos/coords/dec >= -15.0
  and pos/coords/dec <= 20.0]/rms
  |count 8 step 4|)
return <avg_rms>{$s}</avg_rms>
    ]]>
  </query>

</queries>

<injectorMapping>
  <mapping peer="4" stream="stream-0" />
  <mapping peer="1" stream="stream-1" />
  <mapping peer="3" stream="stream-2" />
</injectorMapping>

<queryMapping>
  <mapping peer="2" query="0" install_interval="0" />
  <mapping peer="6" query="1" install_interval="0" />
  <mapping peer="1" query="2" install_interval="0" />
  <mapping peer="2" query="3" install_interval="0" />
  <mapping peer="12" query="4" install_interval="0" />
</queryMapping>

</scenario>
```

D.7 Bypassing Scenario

Filename bypassing.xml

```
<?xml version="1.0" ?>
 <scenario name="GeneratedBenchmark" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
      xmlns="http://streamglobe.net/scenario">

<statistix dbPath="${user.home}/statistiX" reportType="file" />

<graph>
  <vertex vid="0" label="(1)" />
  <vertex vid="1" label="(3)" />
  <vertex vid="2" label="(2)" />
  <vertex vid="3" label="(0)" />
  <edge source="0" target="3" />
  <edge source="1" target="2" />
```

```
  <edge source="2" target="0" />
  <edge source="1" target="0" />
</graph>

<streams>
  <kindDefinition>
    <kind name="file" class="@CONTENT_SERVER_CLASS@" />
  </kindDefinition>
  <stream sid="stream-0" type="file">
    <dtd filename="etc/schemas/vela_nested.dtd" />
    <param name="stream.filename">@FILE_DIR@/vela/vela_demo.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_0@</param>
    <param name="stream.sleep.time">10</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
  <stream sid="stream-1" type="file">
    <dtd filename="etc/schemas/FIRST.dtd" />
    <param name="stream.filename">@FILE_DIR@/first/first-10MB.xml</
        param>
    <param name="stream.server.port">@STREAM_SERVER_PORT_1@</param>
    <param name="stream.sleep.time">10</param>
    <param name="stream.cycle.mode">true</param>
  </stream>
</streams>

<queries>
  <query qid="0">
    <![CDATA[
    display(stream("stream-1"))
    ]]>
  </query>

  <query qid="1">
    <![CDATA[
let $s := avg(stream("stream-1")/first/source
[pos/coords/ra >= 1.0
  and pos/coords/ra <= 140.0
  and pos/coords/dec >= -15.0
  and pos/coords/dec <= 20.0]/rms
  |count 8 step 2|)
return <avg_rms>{$s}</avg_rms>
    ]]>
  </query>

  <query qid="2">
    <![CDATA[
for $p in stream("stream-0")/photons/photon
where $p/coord/cel/ra >= 260
  and $p/coord/cel/ra <= 350
  and $p/coord/cel/dec >= -78
```

```
  and $p/coord/cel/dec <= 20
return
  <vela>
    {$p/coord/cel/ra} {$p/coord/cel/dec}
        {$p/en}
  </vela>
    ]]>
  </query>
</queries>

<injectorMapping>
  <mapping peer="2" stream="stream-0" />
  <mapping peer="2" stream="stream-1" />
</injectorMapping>

<queryMapping>
  <mapping peer="0" query="0" install_interval="0" />
  <mapping peer="0" query="1" install_interval="0" />
  <mapping peer="3" query="2" install_interval="10" />
</queryMapping>

</scenario>
```

Acronyms

AGN active galactic nucleus.

CRM Customer Relationship Management.

DEC declination.

DSMS Data Stream Management System.

DTD Document Type Definition.

ERP Enterprise Resource Planning.

GAVO German Astrophysical Virtual Observatory.

GRB Gamma-ray burst.

GSH Grid Service Handle.

GUI graphical user interface.

LOI luminary of interest.

MPE Max Planck Institute for Extraterrestrial Physics.

OGSA Open Grid Services Architecture.

P2P Peer-To-Peer.

RA right ascension.

RFID Radio Frequency Identifier.

SED spectral energy distribution.

SGG StreamGlobeGui.

SGTP StreamGlobe Transfer Protocol.

XML Extended Markup Language.

Bibliography

[AKL+04] H.-M. Adorf, F. Kerber, G. Lemson, R. Mignani, A. Micol, T. Rauch, and W. Voges. Assembly and classification of spectral energy distributions - a new vo web service. In *Conf. on Astronomical Data Analysis Software and Systems*, Pasadena, CA, USA, November 2004.

[APST05] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4):34–41, April 2005.

[Asc98] B. Aschenbach. Discovery of a young nearby supernova remnant. *Nature*, 396(6707):141–142, November 1998.

[ATE05] The astronomers's telegram. http://www.astronomerstelegram.org, September 2005.

[BDK+03] Ingo Brunkhorst, Hadhami Dharaief, Alfons Kemper, Wolfgang Nejdl, and Christian Wiesner. Distributed queryies and query optimization in schema-based p2p-systems. In *International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, Berlin, Germany, September 2003.

[BKK+01] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. Objectglobe: Ubiquitous query processing on the internet. *The VLDB Journal*, 10(1):48–71, August 2001.

[BSS04] Gert Brettlecker, Heiko Schuldt, and Raimund Schatz. Hyperdatabases for peer-to-peer data stream processing. In *ICWS*, pages 358–, 2004.

[CBA05] Central bureau for astronomical telegrams. http://cfa-www.harvard.edu/iau/cbat.html, September 2005.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman,

F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *Proc. of the Conf. on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[Coh03] Bram Cohen. Incentives build robustness in BitTorrent. http://www.bittorrent.com/bittorrentecon.pdf, May 2003.

[CRA04] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. Gates: A grid-based middleware for processing distributed data streams. In *HPDC*, pages 192–201, 2004.

[dis05] distributed.net: Project RC5. http://www.distributed.net/rc5/, September 2005.

[DKNW04] Hadhami Dhraief, Alfons Kemper, Wolfgang Nejdl, and Christian Wiesner. Processing and optimization of complex queries in schema-based p2p-networks. In *DBISP2P*, pages 31–45, 2004.

[DRF04] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale xml dissemination service. In *Proc. of the Intl. Conf. on Very Large Data Bases* [vld04], pages 612–623.

[Fen05] Bo Feng. Optimierung und Bewertung der Anfrageauswertung auf Datenströmen in P2P Netzwerken. Master's thesis, Technische Universität München, Germany, 2005.

[FKNT02] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, June 2002. http://www.globus.org/research/papers/ogsa.pdf.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. 15(3):200–222, August 2001.

[Fos02] Ian Foster. What is the grid? a three point checklist. *GRIDToday*, July 2002.

[Fou05] The Apache Software Foundation. Apache Ant 1.6.7. http://ant.apache.org, September 2005.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 2003.

[GAV05] German Astrophysical Virtual Observatory. http://g-vo.org, September 2005.

[GCN05] The gamma-ray burst coordinates network. http://gcn.gsfc.nasa.gov, September 2005.

[GRJV95] Erich Gamma, Helm Richard, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GSF+04] Jim Gray, Alexander S. Szalay, Gyorgy Fekete, William O'Mullane, Maria A. Nieto Santisteban, Aniruddha R. Thakar, Gerd Heber, and Arnold H. Rots. There goes the neighborhood: Relational algebra for spatial data search. Technical Report MSR-TR-2004-32, Microsoft Research, Microsoft Cooperation, Redmond, WA, USA, April 2004.

[HT99] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Häu05] Franz Häuslschmid. Infrastrukturen und verfahren zur adaptiven anfragebearbeitung und optimierung in datenstrom-management-systemen. Master's thesis, Universität Passau, Germany, 2005.

[Ill79] Valierie Illingworth, editor. *The Macmillan Dictionary of Astronomy*. MacMillan Refernce Books, 1979.

[JXT05] Project JXTA. http://www.jxta.org, September 2005.

[KE01] Alfons Kemper and André Eickler. *Datenbanksysteme - Eine Einführung, 4. Auflage*. Oldenbourg, 2001.

[KSH+04] R. Kuntschke, B. Stegmaier, F. Häuslschmid, A. Reiser, A. Kemper, H.-M. Adorf, H. Enke, G. Lemson, and W. Voges. Datenstrom-Management für e-Science mit StreamGlobe. *Datenbank-Spektrum*, 4(11):14–22, November 2004.

[KSK05] Richard Kuntschke, Bernhard Stegmaier, and Alfons Kemper. Data stream sharing. Technical Report TUM-I0504, Technische Universität München, 2005. to be published.

[KSKR05] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser. Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures. In *Proc. of the Intl. Conf. on Very Large Data Bases*, pages 1259–1262, Trondheim, Norway, August 2005.

[KSSS04] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization on structured data streams. In *Proc. of the Intl. Conf. on Very Large Data Bases* [vld04], pages 228–239.

[KW01] Alfons Kemper and Christian Wiesner. Hyperqueries: Dynamic distributed query processing on the internet. In *VLDB*, pages 551–560, 2001.

[KW05] Alfons Kemper and Christian Wiesner. Building scalable electronic market places using hyperquery-based distributed query processing. *World Wide Web*, 8(1):27–60, 2005.

[NSS03] Wolfgang Nejdl, Wolf Siberski, and Michael Sintek. Design issues and challenges for rdf- and schema-based peer-to-peer systems. *SIGMOD Record*, 32(3):41–46, 2003.

[NST03] Wee Siong Ng, Yanfeng Shu, and Wee Hyong Tok. Efficient distributed continous query processing using peers. Technical Report NUS-CS01-03, National Universtity of Singapore, 2003.

[Pla05] PlanetLab. http://planet-lab.org, September 2005.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, *Middleware*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[SET05] SETI@home. http://setiathome.ssl.berkeley.edu, September 2005.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[SSDN02] Mario T. Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. Hypercup - hypercubes, ontologies, and efficient search on peer-to-peer networks. In *AP2PC*, pages 112–124, 2002.

[Sun05] Sun Microsystems Inc. Java™ 2 Platform Standard Edition 5.0. http://java.sun.com/j2se/1.5.0/, September 2005.

[SWA+05] Rob Seaman, Roy Williams, Alasdair Allan, Scott Barthelmy, Joshua Bloom, Frederic hessman, Szabolcs Marka, Arnold Rots, Kate Scholberg, Chris Stoughton, Tom Bestrand, Robert White, and Przemyslaw Wozniak. Sky event reporting metadata (VOEvent. http://www.ivoa.net/internal/IVOA/IvoaVOEvent/VOEvent-v1.0.html, September 2005.

[TB05] Niklas Therning and Lars Bengtsson. Jalapeno: secentralized grid computing using peer-to-peer technology. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 59–65, New York, NY, USA, 2005. ACM Press.

[TMM05] Henry S. Thompson, Murray Maloney, and Noah Mendelsohn. Xml schema part 1: Structures second edition. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/, September 2005.

[VAB+99] W. Voges, B. Aschenbach, Th. Boller, H. Bräuninger, U. Briel, W. Burkert, K. Bennerl, J. Englhauser, R. Gruber, F. Haberl, G. Hartner, G. Hasinger, M. Kürster, E. Pfeffermann, W. Pietsch, P. Predehl, C. Rosso, J.H.M.M. Schmitt, J. Trümper, and H.U. Zimmermann. The rosat all-sky survey bright source catalogue. *Astronomy and Astrophysics*, 349(2):389–405, July 1999.

[vld04] *Proc. of the Intl. Conf. on Very Large Data Bases*, Toronto, Canada, August 2004.

[Wik05a] Wikipedia, the free encyclopedia. http://en.www.wikipedia.org, September 2005.

[Wik05b] Wikipedia. Stellar classification. http://en.wikipedia.org/wiki/Stellar_classification, September 2005.

Acknowledgements

At the end of my thesis, I would like to take the opportunity to thank everyone who made this work possible.

Especially, *Richard Kuntschke* for great guidance during my thesis, for keeping me focused and giving insightful comments. Thanks to *Angelika Reiser* for her helpful advice to bring this "raw diamond" into shape and many fruitful discussions.

I am grateful to *Prof. Alfons Kemper*, for giving me the opportunity "to reach for the stars" and for encouraging me to commit myself to this topic.

For opening the world of astrophysics, I would like thank the people at the MPE, especially *Wolfgang Voges*,*Gerard Lemson*, and *Hans-Martin Adorf*. They provided the astrophysical catalogs, astrophysical images, Java libraries, and patience to explain some of the essentials several times.

Working in a team on StreamGlobe or StarGlobe together with *Franz Häuslschmid*, *Bo Feng*, and *Sebastian Huber* has been a great experience, especially during some of those surrealistic white board sessions.

Thanks to my proofreaders, *Tobias Brandl*, *Thomas Harrer*, *Christian Schneider*, *Stefan Schöffmann*, and *Marcus Simon* for their help and constructive criticism.

Finally, I warmly thank my parents *Elisabeth and Hartmut Scholl* and *Nina Hentschel* for their ideas, patience, encouragements, advice, and love during all stages of this thesis and my studies.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Diplomarbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle wörtlich oder sinngemäß übernommenen Ausführungen wurden als solche gekennzeichnet. Weiterhin erkläre ich, dass ich diese Arbeit in gleicher oder ähnlicher Form nicht bereits einer anderen Prüfungsbehörde vorgelegt habe.

Passau, den 19. September 2005 .....

Tobias Scholl