COMPUTING SCIENCE

A Toolkit for Constructing Distributed Object-Oriented Metainformation Systems

A. Calsavara and S.K. Shrivastava

TECHNICAL REPORT SERIES

No. 557 November, 1996
A Toolkit for Constructing Distributed Object-Oriented Metainformation Systems

A. Calsavara and S.K. Shrivastava

Abstract

Stabilis is a programming tool for the construction of distributed object-oriented metainformation systems whose main purpose is to deliver powerful and reliable object query services in large-scale distributed environments. Metainformation consists of a catalogue with an object-oriented structural description of information and corresponding indices to information objects. This permits queries to be formulated in a highly structured fashion, thus exploiting semantic knowledge about information. Information objects are external to Stabilis and can be any entity that contains information, including the resources available on the Internet. The notion of views is also supported in order to permit organising the information space according to user needs, such as topic-specific information. Transactional access is employed to obtain consistency, and replication is employed to obtain high availability and scalability. Stabilis is implemented as an extensible C++ class library atop a distributed transaction facility named Arjuna. We describe Stabilis using as an example a system for querying about bibliographical references which has been constructed as a demonstration application.
# Bibliographical details

**CALSAVARA, Alcides**

A Toolkit for Constructing Distributed Object-Oriented Metainformation Systems  
[By] A. Calsavara and S.K. Shrivastava  
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 557)

---

**Added entries**

UNIVERSITY OF NEWCASTLE UPON TYNE.  
SHRIVASTAVA, Santosh Kumar

---

**Abstract**

**Stabilis** is a programming tool for the construction of distributed object-oriented metainformation systems whose main purpose is to deliver powerful and reliable object query services in large-scale distributed environments. Metainformation consists of a catalogue with an object-oriented structural description of information and corresponding indices to information objects. This permits queries to be formulated in a highly structured fashion, thus exploiting semantic knowledge about information. Information objects are external to Stabilis and can be any entity that contains information, including the resources available on the Internet. The notion of views is also supported in order to permit organising the information space according to user needs, such as topic-specific information. Transactional access is employed to obtain consistency, and replication is employed to obtain high availability and scalability. Stabilis is implemented as an extensible C++ class library atop a distributed transaction facility named Arjuna. We describe Stabilis using as an example a system for querying about bibliographical references which has been constructed as a demonstration application.

---

**About the author**

Dr. A. Calsavara received a Ph.D. in Computing Science from the University of Newcastle upon Tyne, UK, in 1996. His current research addresses the development of fault-tolerant distributed applications, object-oriented databases and the construction of meta-information systems in large-scale distributed environments. He is currently with the Departamento de Informatica, Pontificia Universidade Catolica do Parana, Brazil.

S.K. Shrivastava joined the Department of Computing Science in August 1975, where he is a Professor.

---

**Suggested keywords**

<table>
<thead>
<tr>
<th>INFORMATION DISCOVERY</th>
<th>INFORMATION RETRIEVAL</th>
<th>METADATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>METAINFORMATION</td>
<td>OBJECT-ORIENTED DATABASE</td>
<td>OBJECT QUERY</td>
</tr>
</tbody>
</table>

**Suggested classmarks (primary classmark underlined)**

Dewey (18th): 001.64404 001.6442 029.753

U.D.C. 519.687 681.322.06 651.838.8
A toolkit for constructing distributed object-oriented metainformation systems*

A. Calsavara
S. K. Shrivastava

University of Newcastle upon Tyne, UK
Department of Computing Science

{a.calsavara,santosh.shrivastava}@newcastle.ac.uk

May 24, 1996

Abstract

Stabilis is a programming tool for the construction of distributed object-oriented metainformation systems whose main purpose is to deliver powerful and reliable object query services in large-scale distributed environments. Metainformation consists of a catalogue with an object-oriented structural description of information and corresponding indices to information objects. This permits queries to be formulated in a highly structured fashion, thus exploiting semantic knowledge about information. Information objects are external to Stabilis and can be any entity that contains information, including the resources available on the Internet. The notion of views is also supported in order to permit organising the information space according to user needs, such as topic-specific information. Transactional access is employed to obtain consistency, and replication is employed to obtain high availability and scalability. Stabilis is implemented as an extensible C++ class library atop a distributed transaction facility named Arjuna. We describe Stabilis using as an example a system for querying about bibliographical references which has been constructed as a demonstration application.

Keywords: metainformation, metadata, information discovery, information retrieval, object query, object-oriented database.

*This work has been supported by grants from UK Engineering and Physical Sciences Research Council, and from CNPq-Brasilia/Brazil.
1 Introduction

In this paper we present a toolkit for constructing metainformation systems\(^1\) that purpose to provide an object-oriented interface to information contained in network resources in large-scale distributed environments, such as the Internet.\(^2\) The main reason for constructing object-oriented metainformation systems is that the query services currently available for large-scale distributed environments provide little support for structured information retrieval. Complementary to searching engines, traditionally used for information discovery in global networks, which aim at ill-structured information, object-oriented metainformation systems aim at well-structured information and, therefore, they are intended to be used in conjunction with searching engines. For simplicity, in the remaining of this paper, we call object-oriented metainformation systems Stabilis brokers.

In general, the tools for information discovery currently available conform to the scheme illustrated in Figure 1. Users submit keyword-based queries to searching engines, obtaining references to network resources, and subsequently access the corresponding network resources. Searching engines maintain index information up-to-date by frequently collecting keywords from network resources.

![Figure 1: Scheme normally employed by information discovery tools](image)

Certainly, the approach taken by searching engines, where users express syntactic knowledge about information, is appropriate for the cases where either the information presents no

\(^1\)A metainformation system is a system that manages some information about an information base and normally delivers very specific services.

\(^2\)Typical network resources include file systems, databases and hypertext systems such as the World Wide Web.
structure or has a structure that is unknown to users. However, a problem that frequently arises in keyword-based information systems is the large size of results containing non-relevant hits. For a simple example, let us consider that one wants to retrieve information about books on beetle, the insect, but only the books that are suitable for children aged between 7 and 10, and written in English. Firstly there is no way to specify the age range or the language, and secondly, if only \textit{beetle} is specified as a keyword, it is likely to obtain as an answer a lot of information about people that has this keyword in their names, information related to the car that has been so nicknamed, and perhaps information related to the musical band whose name is an approximate match. Of course, one may suggest that the problem of category mismatch pointed by this example could be simply solved by providing \textit{book} as the second keyword in the query. However, seldom will any keyword-based query service keep this word to refer to a potentially countless number of books — it certainly will refer to other things related to book, such as book shops, publishers, and so on. Although simple, this example reveals the weaknesses of a strict keyword-based query service: query results can be too large to be transmitted, stored and treated, at reasonable cost and time.

On the other hand, a significant part of the network resources in the global information base has well-defined structure and well-defined interrelationship paths which, if properly exploited, would permit users to formulate queries where they express \textit{semantic} knowledge about information, thereby contributing to increase the rate of relevant hits. Object-oriented data modelling, in particular, represents an end-point in the evolution of data modelling and is advocated to be an effective approach to the representation of real-world complex entities and their relationships. This suggests that information contained in network resources, including the links between these resources, would perfectly be modelled using object-oriented concepts. Thus, a schema (a collection of classes organised in hierarchies), devised to represent the structure and the relationships of information contained in a collection of network resources, would allow users to formulate queries in a highly-structured fashion. Also, users would be able to navigate through information by navigating through objects, and perform operations on network resources by calling object methods. Furthermore, many distributed systems, including operating systems and platforms for distributed programming, advocate the use of object orientation as an adequate framework for their internal structuring and as a powerful abstraction at the user interface level. In particular, the \textit{object and action model of computation} is a widely accepted approach to reliable distributed computing. Examples of systems based on this model are Arjuna [20] and Camelot [9].

Our objective is to contribute towards the efficiency, effectiveness and reliability of information systems in large-scale distributed environments. Information systems would gain in efficiency through Stabilis brokers because query results would present a higher rate of relevant hits, when compared with searching engines. The gain in effectiveness would come from
the power of object-oriented data modelling; an object-oriented view of information would provide a propitious abstraction for developing applications to manipulate information that has complex structure and relationships. Finally, the gain in reliability would be achieved due to the use of object-oriented techniques which are well-established in modern distributed systems; transactional access would provide for consistency, while object replication for high availability and scalability. The basic use of Stabilis brokers can be summarised as follows. Client programs submit object-oriented queries to Stabilis brokers obtaining a set of objects which are instances of classes as a result. Then, the resulting objects are used for the following purposes:

1. Retrieve information resources: objects may contain references to information resources.

2. Preview of information resources: objects may contain summaries of information resources.

3. Navigate to related objects: Stabilis brokers resolve object references and return the resulting objects.

4. Perform operations on information: objects may provide methods which can manipulate the corresponding summaries and information resources.

5. Create and modify objects: in addition to objects extracted from information resources, Stabilis brokers may maintain objects created by clients, thereby behaving as information resources as well.

The remainder of this paper is organised as follows. Section 2 defines a model for structuring and querying information according to object-oriented programming concepts. Section 3 discusses the features required from Stabilis brokers and outlines the architecture of a toolkit for their construction. Section 4 further explains how Stabilis brokers are structured in terms of components, query resolution and distribution management. Section 5 describes the class library that implements the toolkit for the construction of Stabilis brokers. Section 6 discusses related work. Section 7 provides conclusions and further research work.

2 Information Modelling

Information contained in network resources is modelled by employing the notions of encapsulation, identity, classification, inheritance (generalisation/specialisation) and relationship found in object-oriented programming and in database systems. The purpose of modelling is to define a schema, a collection of classes which describe object properties and are arranged in a certain way to ensure that the objects which belong to these classes compose a consistent
database and, therefore, can be properly manipulated. An example of network resource is shown in Figure 2, and a schema that models information contained in this resource (bibliographical references in BibTeX format [14]) is shown in Figure 3. This example is explained in the following discussion.

2.1 Object Data Model

The properties of an object are captured by a state and a corresponding interface, similarly to an abstract data type. An object state consists of values of atomic types, such as string and integer, and references to other objects. These values are said to be object attributes, and these references are said to represent object relationships. Every object is provided with an identity, a unique name to permit objects to be referred to unambiguously. An object interface consists of functions that have exclusive access to an object state, i.e., an object interface encapsulates an object state. These functions are referred to as methods.

Objects are classified according to their properties; a class stands for an abstract data type and for a subset of objects that have some properties in common — an object which is member of such a subset is said to be an instance of the respective class. The result of a recurrent classification is a directed tree referred to as a class hierarchy, where each node corresponds to a class and each arc (between two classes) specifies generalisation/specialisation. The direction of an arc is from a superclass to a subclass: the superclass defines more general properties, while a subclass defines more specific properties. Thus, instances of a subclass have the properties defined by that subclass and the properties defined by its superclass: an object state consists of the attributes and relationships defined by its most specific class, and all the attributes and relationships defined by its superclasses as well, while an object interface is composed by the methods defined by its most specific class and all the methods defined by its superclasses. Conceptually, a subclass inherits the properties defined by its superclass. Moreover, an instance of a subclass is also an instance of the respective superclass. As a consequence, an instance of any subclass can appear where an instance of its superclass is expected — a property normally called substitutability. In the example, a class hierarchy is defined by the classes Reference, Publication, Article, Journal and Book, meaning that:

- **Reference** is superclass of Article and Publication. Reference defines the attribute title which is common to Article and Publication.

- **Article** defines the attribute pages. So, an instance of Article has the attribute title inherited from Reference and the attribute pages.

- **Publication** is superclass of Journal and Book. Publication defines the attributes year and publisher which are common to Journal and Book. So, an instance of Publication
has the attribute *title* inherited from *Reference*, and the attributes *year* and *publisher*.

- *Journal* defines the attribute *month*. So, an instance of *Journal* has the attributes *title*, *year* and *publisher* inherited from *Publication*, and the specific attribute *month*.

- *Book* defines no attribute. So, an instance of *Book* has the same attributes as instances of its superclass, i.e., the attributes *title*, *year* and *publisher* inherited from *Publication*.

- Instances of *Article*, *Journal* and *Book* are instances of *Reference*. So, either an instance of *Article*, *Journal* or *Book* can substitute an instance of *Reference*.

**Relationships** between objects can have different semantics and are termed accordingly. An *association* between two classes means that their instances have a conceptual connection: each class has a *role* in the association, and the range of instances of each class participating in the association defines the *multiplicity* of each class. In the example, the classes *Individual* and *Article* have an association between them to represent authorship, meaning that:

- The role of *Individual* in the association is *Author*, while the role of *Article* in the association is *Article* (there is no other specification).

- The multiplicity of *Individual* is \([1,n]\) (an instance of *Article* is associated to one or many instances of *Individual*, i.e., an article has one or many authors), while the multiplicity of *Article* is \([0,n]\) (an instance of *Individual* is associated to none, one or many instances of *Article*, i.e., an individual is author of any number of articles).

An *aggregation* between two classes means that instances of one class, the *component class*, are part of instances of another class, the *aggregate class*. In the case where component objects are *physically* part of an aggregate object the aggregation is termed *tight aggregation*. Otherwise, when component objects are *conceptually* part of aggregate objects the aggregation is termed *loose aggregation*. In the example there is a tight aggregation between *Journal* and *Article*, and a loose aggregation between *Library* and *Publication*, meaning that:

- An instance of *Article* is physically part of an instance of *Journal*, while an instance of *Journal* is an aggregate of any number of instances of *Article*.

- An object that is an instance of *Publication* is conceptually part of any number of instances of *Library*, while an instance of *Library* is an aggregate of any number of instances of *Publication*. 
Figure 2: Example of a network resource: a BibTeX file

Figure 3: A schema that partially models information contained in BibTeX files
2.2 Object Query

An object query, or simply query, is a declarative specification of objects according to their properties: a query permits whole objects to be retrieved by specifying just some of their properties. The result of a query is the set of objects whose properties are in conformity with predicates expressed in terms of classes, attributes, relationships and methods defined by a schema. A query is schema conservative: neither objects nor classes are created as a result of a query. Thus, a set of objects obtained as a query result is composed of objects which are existing instances of existing classes. Table 1 illustrates the basic constructs of the query language, with some queries formulated against the schema in the example. The queries are expressed in both natural and algebraic languages. These basic constructs can be combined to formulate more complex queries, as defined by the grammar presented in Appendix A.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Query expressed in natural and algebraic languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>exact match</td>
<td>Book titled <em>Object-Oriented Software Construction</em></td>
</tr>
<tr>
<td></td>
<td><em>Book(title = 'Object – Oriented Software Construction')</em></td>
</tr>
<tr>
<td>approximate match</td>
<td>Books whose titles contain the word <em>object</em> (case insensitive)</td>
</tr>
<tr>
<td></td>
<td><em>Book(title % 'object')</em></td>
</tr>
<tr>
<td>Boolean operator</td>
<td>Books whose titles contain either the word <em>object</em> or the word <em>software</em></td>
</tr>
<tr>
<td>relational operator</td>
<td>*Book(title % 'object'</td>
</tr>
<tr>
<td>relational operator</td>
<td><em>Book(year &gt; 1980 &amp;&amp; year &lt;= 1990)</em></td>
</tr>
<tr>
<td>associative access</td>
<td>Books whose authors have surname <em>Meyer</em></td>
</tr>
<tr>
<td></td>
<td><em>Book(Author(surname = 'Meyer'))</em></td>
</tr>
<tr>
<td>superclass access</td>
<td>References (books, journals or articles) whose titles contain the word <em>object</em></td>
</tr>
<tr>
<td></td>
<td><em>Reference(title % 'object')</em></td>
</tr>
<tr>
<td>attribute casting</td>
<td>References which are either publications published in 1988 or articles whose page numbers are 16 to 33</td>
</tr>
<tr>
<td></td>
<td>*Reference([Publication]year = 1988</td>
</tr>
<tr>
<td>role casting</td>
<td>Libraries that contain books whose titles contain the word <em>modeling</em></td>
</tr>
<tr>
<td></td>
<td><em>Library([Book]Publication(title % 'modeling'))</em></td>
</tr>
</tbody>
</table>

Table 1: Examples of use of the query language basic constructs
3 Architecture Overview

3.1 Rationale

Stabilis brokers conform to the scheme discussed in Section 1 and illustrated in Figure 1. The architecture of the environment where Stabilis brokers operate is represented by the diagram in Figure 4. The metainformation maintained by a Stabilis broker consists of schemas and corresponding indices to information objects in order to resolve queries. An information object in its natural representation, i.e., within a network resource, is denominated primary object. An entity denominated object representative is introduced as a front-end to the primary object so that together they implement an information object. More specifically:

1. The primary object is only required to hold state, i.e., data. However, since queries can invoke methods, the information object is required to be a state encapsulated by an interface that supports method invocation. Moreover, data encapsulation eases the development of programs because it permits the information base to be uniformly manipulated as an object-oriented database: client programs obtain information objects as the result of queries and then invoke their methods to manipulate them, without concerning with their representation. For example, an information object can have a method that prints its attributes, or a method that carries a complex computation. For these reasons, the object representative implements the information object interface.

2. The primary object is not required to contain explicit information about its relationships with other primary objects. However, the information object is required to support navigational access, i.e., relationships can be traversed in order to examine related information objects. For this reason, the object representative needs to maintain a representation of the information object relationships.

3. The primary object is not required to be an entity that is concurrently, remotely and reliably accessible from programs. However, information objects are required to be accessible in such a way. For this reason, the representative object provides the respective abstractions and mechanisms.

The object representative contains a summary of the primary object: its state contains a copy of the part of the primary object that is necessary and sufficient to implement its interface and a representation of its relationships. Thus, the state of the object representative contains at least the indexed attributes and the indexed relationships, i.e., attributes which are relevant for queries and the relationships which are interesting for navigation. For example, if it is defined for the class Individual in Figure 3 that the attribute surname is relevant for query
and that the attribute `forenames` is not, then the representative object will contain only the attribute `surname`. But it may be necessary for a method to access attributes or relationships which are found only in the primary object. In this case, the object representative has to fetch the missing part from the primary object.

As a consequence of the introduction of the object representative, indices make reference to them rather than directly to primary objects. This is indicated in Figure 4 by the dashed arrows. The reference from an index to a primary object is represented by the name that uniquely identifies the information object. Such a name is not visible to end-users; it is rather system generated when the object representative is created, and used for the purpose of physical address independence. The format of the reference from object representative to primary object varies according to network resources and primary object representation. In the case where the network resource is a BibTeX file, for example, this reference may be the file name concatenated with the unique tag associated to each file entry, such as `/home/smith/bibtex/obj.bib+Meyer88`. As another example, if the network resource is a World Wide Web page then the reference from the object representative is the Uniform Resource Locator (URL) of the page, such as `http://www.newcastle.edu.au/`. Since this is an application-specific aspect, it is no further discussed in this paper.

The function of mapping information objects from their natural representation to a schema is implemented by entities denominated **collectors**. They are responsible for no-
tifying Stabilis brokers about creation, modification and deletion of primary objects for the purpose of index update. This is accomplished by periodically interrogating network resources about modifications on primary objects and then taking the proper action on representative objects. Collectors benefit from the encapsulation of index update concerns provided by representative objects, i.e., collectors simply manipulate object representatives, as shown in Figure 4, and have index information automatically updated.

3.2 Views and Contexts

A view is a portion of a schema. A view is defined by selecting a subset of the classes from a schema. Such a subset honours the class hierarchy and relationships defined in the schema in order to ensure that the encapsulation principle is maintained, i.e., the selection of a certain class implies the selection of all its superclasses and all the related classes, recursively. A schema can have any number of associated views, and intersections between them can happen freely. A view defines the scope of a query, i.e., a query is resolved against a view rather than against the whole schema. As a consequence, only the information objects which are instances of the classes in the view are accessible — we can say that a view is a logical container of information objects. For this reason, the information space can be organised by properly defining views. More specifically, views serve to the following purposes:

Customisation: Views aid in coping with information overload as they permit users to select information according to their specific needs. For example, users can create topic-specific views of information.

Security: Views are a means of restricting access to information, thus facilitating the organisation of user groups.

Performance: Views permit to save on time and costs (especially processing) since they constrain the scope of query resolution.

Reliability: Views are the units of replication of metainformation, which means that their definition must take certain physical systems characteristics into account, such as network topology and probability of network partition and node failure.

Administration: Views are a means of decentralising administration in large-scale distributed environments, i.e., they permit to define administrative domains.

Schemas and views themselves constitute an information base and, consequently, they need proper management, such as controlling access rights to views. For this reason, a schema and its corresponding views are maintained by a logical container called context: a context
contains a collection of schemas and all their associated views. Thus, systems administration programs interact with a context in order to create, update and delete schemas and views. Also, collectors and client programs interact with a context in order to get access to views.

3.3 Operational Requirements

Stabilis brokers are built to satisfy a set of interrelated features which are normally required in large-scale distributed environments. These features include concurrence control, remote access, timely access, consistency, fault tolerance, high availability, decentralisation, heterogeneity management, portability and scalability. The issues related to these features are numerous and have implications in practically all the components of a Stabilis broker. Thus, they are addressed at the design level and at the implementation level in Sections 4 and 5, respectively.

4 Design

4.1 Stabilis Brokers Components and Query Resolution

The physical schema of a Stabilis broker is similar to an inverted file. This is illustrated in Figures 5 and 6, where the dashed-line-style triangle surrounds the index part and the circle surrounds the data part, i.e., information objects. Figure 5 shows how attributes are indexed, while Figure 6 shows how relationships are indexed. From top to bottom, the hexagon surrounds metainformation corresponding to schemas, the solid-line-style triangle surrounds metainformation corresponding to index, the rectangle surrounds the data maintained by a naming system, and the circle surrounds the information objects. A naming system is necessary in order to resolve references from indices to objects: names used by indices are symbolic addresses which are mapped to physical addresses. A query is resolved by translating an algebraic query expression into a tree representation and then reducing the tree using the metainformation.

The internal organisation of indices can be better understood through an example. Figure 7 illustrates how information objects are structured and what index information is maintained about them. The two information objects in the example correspond to the first entry in the BibTeX file shown in Figure 2. For simplicity, in the index diagrams only the entries corresponding to the objects in the example are shown. The information objects are an instance of class Book, named A, and an instance of class Individual, named B. The indexed attributes are the title of a book and the surname of an individual, and the indexed relationship is the association of authorship between books and individuals, in both directions.
Figure 5: Physical schema for metainformation on attributes

Figure 6: Physical schema for metainformation on relationships
Sets of names are represented by names within braces, such as \( \{A\} \) or \( \{X,Y,Z\} \). This is used to represent information about relationship within each object and also to represent references from indices to objects. For example, the relationship \textit{Author} in the instance of \textit{Book} is represented by the set \( \{B\} \) to indicate that the object with name \( B \) corresponds to an individual that is author of the book. This same relationship is also represented in the relationship index for class \textit{Individual} and role \textit{BookAsAuthor}, but in the reverse order.

![Diagram of information objects and respective indices](image)

Figure 7: Example of information objects and respective indices

### 4.2 Distributed Metainformation Access and Replication

Figure 8 illustrates how distributed metainformation is accessed by programs (collectors and client programs). The model of distribution is client-server processes with communication through Remote Procedure Call (RPC). Programs directly access views and representative objects, as queries are formulated against views to retrieve object representatives, and indirectly access indices, as only views access them to resolve queries. Thus, views are expected to be accessed very frequently and for read-only operations (query resolution does not modify metainformation). Moreover, views are expected to suffer very few modifications along their lives. Another expected pattern is that clients typically access a few different views. For these reasons, programs maintain views in cache — once a program begins, the state of a view is copied to cache and then the view methods are (locally) invoked. Indices, on the other hand,
are expected to be frequently updated, and programs are expected to access many different indices. For these reasons, indices are not copied to programs cache. Object representatives are expected to be relatively small, programs are expected to access many of them, they can be updated when collectors run, and they are expected to provide methods which are very often invoked by programs. For these reasons, object representative states are locked and then copied to programs cache on demand, i.e., for every transaction a program initiates that invokes object representative methods. If an invoked method modifies the cache then the update is propagated back to the state and the state is freed from the lock when the transaction is finished. More details about the management of cache and object state can be found in our previous work [7].

Figure 8: Distributed metainformation access scheme

Views, indices and representative objects are replicated for the purposes of high availability and scalability: replication prevents the existence of a single point of failure, and at the same time prevents the existence of bottlenecks in the system. Replication together with adequate data structures for the metainformation provide scalability to the query service in relation to response time, i.e., query resolution boundness. Consistency is accomplished by providing solely transactional access to metainformation and object representatives.

5 Implementation

Stabilis is a toolkit for constructing brokers that satisfy a set of interrelated features which are normally required in large-scale distributed environments, including concurrence control, remote access, consistency, fault tolerance, high availability, portability and scalability. Stabilis is implemented as an extensible class library atop the Arjuna system [20], and benefits
from it to provide some of the features. C++ is the only programming language used in the implementation and also is the language at the programming interface: query statements are embedded in programs as ordinary C++ quoted strings and interpreted at run-time. As a consequence, neither language extension nor special compilers are necessary, thereby contributing to systems portability.

5.1 Underlying System: Arjuna

Arjuna is a distributed transaction facility; it consists of a set of tools that supports the object and action model of computation, a widely accepted approach to reliable distributed computing. In this model, programs consist of interacting objects, where every interaction happens within an atomic action, a programming abstraction that ensures serialisability, failure atomicity and permanence of effect. Coherence is accomplished by the enforcement of encapsulation; the state of the system is maintained solely by objects, and the state of each object is manipulated only by associated access methods, which, by definition, are the units of interaction. By ensuring that objects are recoverable and only manipulated within an atomic action, it can be guaranteed that the integrity of objects — hence the integrity of the system — is maintained in the presence of failures such as node crash and message loss.

The main system facilities include object storage (transparent persistence management), nested atomic actions (transparent distributed transaction management), remote object access (transparent remote method invocation using RPC), concurrence control, crash recovery and object replication. The current version of Arjuna is implemented as a standard C++ class library.

5.2 Class Library and Utilities

The main modules of the StabiliS class library are the following:

Object: This module contains a class named Object, for individual manipulation of object representatives, and a class named ObjectSet, for manipulation of a set of object representatives. All classes are rooted at the class Object in order to inherited operations for atomicity, caching, relationship management (creation, deletion and navigation), automatic index update and a query interface.

Query: This module provides classes for query resolution. These classes are intermediary between the classes Object and ObjectSet and the classes provided by the module Meta. Also, this module provides classes for the automatic generation of interactive query interpreters tailored to schemas.
Meta: This module provides a collection of classes that implement a schema for the information objects which represent schemas, including user-defined ones and itself. That is, the classes in this module correspond to the classes of an object-oriented database that keep information about classes (including attributes and methods) and their relationships. This schema, its classes and the instances of these classes are respectively denominated metamodel, metaclasses and metaobjects. This database exists internally to every Stabilis broker and is used by the module Query in order to resolve queries. For this reason, metaobjects of the classes Attribute and Relationship from this module keep references to instances of classes provided by the module Index. Other purposes of this module include automatic generation of code for the classes of schemas, and the generation and manipulation of views. Moreover, the reflexive nature of this database, i.e, the ability to represent itself, permits metaobjects to be queried and manipulated as ordinary objects, in a uniform way — an interactive query interpreter is provided for this purpose. More details about this this module can be found in [8].

Index: This module provides the classes that implement indices for attributes and relationships. Instances of these classes are referred to by instances of classes from the module Meta, and they keep the names of the instances of schemas. Names are implemented as unique identifiers (UIDs), provided by the Arjuna library. Currently, indexed attributes must be either string or integer type. No type restriction, however, is imposed to non-indexed attributes.

NameServer: This module provides a simple naming service for the translation of UID into object address in the format expected by the Arjuna object store management subsystem. It is also responsible for mapping user-defined names of contexts into their UIDs.

5.3 Programming Interface

Object representatives are created, retrieved, modified and deleted from C++ programs. The C++ code that follows are fragments of collectors and client programs. The first code fragment is from a collector program which has been automatically generated for a BibTex file. Firstly, an instance of context named ComputingDepartment is retrieved. — this name is resolved by the name server provided by Stabilis. Secondly, an instance of view named BibliographicalReferences is retrieved — this name is resolved by the context. Next, using the view, two objects are created: an instance of Book and an instance of Individual, as shown in Figure 7. Finally, the two instances are related: the instance of Individual is Author of the instance of Book. For simplicity, the part of the code related to exception handling is not shown.
The next code fragment is from a client program that retrieves some objects after having retrieved a context and a view. Firstly, an instance of Book whose title contains the word software is retrieved. In this case, if more than one instance satisfy the query, then one of them is chosen at random. Next, using the class ObjectSet, all instances that satisfy the same query are retrieved.

5.4 Demonstration Application and Performance

We have fully implemented a Stabilis broker in order to validate Stabilis. The implemented metainformation system supports a schema that models bibliographical references based on all types of entries for bibliography citation defined for \texttt{BibTeX}. The model contains 24 classes, with class hierarchies of depth up to 5, and several associations and aggregations. A partial view of the model is shown in Figure 3 and has been used as example in this paper. To feed Stabilis with bibliographical information, we developed a collector program to extract bibliographical data from files formatted in accordance with the \texttt{BibTeX} syntax. The output of this collector program is a C++ program that creates the corresponding objects. A graphical interface for a partial view of the schema is shown in Appendix B. Some performance figures of the implemented Stabilis broker containing approximately 1,000 information objects is shown in Table 2. The Stabilis broker runs distributed over a set of workstations connected by an Ethernet LAN. The query times include the following: parse the query, transmit subqueries to indices, search the indices, send partial results (UIDs) back to client, and merge the partial results. The retrieve times are average times for transferring objects from remote object stores to client caches. These retrieve times vary according to object size (which depends on attributes and relationships) — their average size is approximately 1 Kbyte.
<table>
<thead>
<tr>
<th>Query expression</th>
<th>Objects</th>
<th>Query time (ms)</th>
<th>Retrieve time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article(title % 'object')</td>
<td>33</td>
<td>216</td>
<td>67</td>
</tr>
<tr>
<td>Journal(title % 'comput')</td>
<td>35</td>
<td>125</td>
<td>120</td>
</tr>
<tr>
<td>Journal(year &gt; 1980)</td>
<td>47</td>
<td>107</td>
<td>120</td>
</tr>
<tr>
<td>Journal(title % 'comput' &amp;&amp; year &gt; 1980)</td>
<td>34</td>
<td>181</td>
<td>128</td>
</tr>
<tr>
<td>Book(Editor(surname % 'e'))</td>
<td>20</td>
<td>188</td>
<td>141</td>
</tr>
</tbody>
</table>

Table 2: Performance of the Stabilis broker for bibliographical references

6 Related Work

The previous research we have built upon basically includes work on:

Information discovery tools: Many tools\(^3\) have recently been developed and, in fact, are available on the Internet. However, to the best of our knowledge, none of these tools supports object queries, i.e., queries based on object-oriented concepts. Some, usually denominated searching engines, take an approach based on keywords: they index information objects which are either plain text or have a part (or a name) from which keywords can be extracted. Examples of these tools include WAIS [13], which supports full-content indexing, Archie [11], which indexes keywords extracted from names, and Harvest [3], which uses summary-content indexing. Surveys on these and similar tools can be found in [18] and [16].

These tools are normally employed in combination with hypertext systems, such as the World Wide Web. Hypertext and keyword-based search do not capture information semantics; objects are retrieved through iterative refinement steps, where information is navigated, rather than retrieved using a precise designation. Typically, network resources are found through hypertext link traversal and keyword-based search specification, possibly in the form of a Boolean combination of terms. Information objects are successively retrieved and then examined in order to select the next object. Such a selection is done either by traversing a hypertext link present in the currently selected object or by formulating a new keyword-based query. Thus, in order to express their knowledge about the information they want to retrieve, users are always forced to proceed by a sequence of refining steps in conformance with the structure defined by object links.

Although Stabilis and Harvest differ significantly with respect to the type of queries they support, we can observe some comparable features between them, including:

- Harvest topic-based brokers are equivalent to Stabilis views: both aim at coping with

\(^3\)These tools are also referred to as networked resource retrieval tools in the literature.
information overload and diversity to provide for scalability.

- Harvest summary objects are equivalent to Stabilis representative objects.

- Harvest permits classification of documents according to a predefined collection of types and corresponding attributes. Stabilis goes further by permitting arbitrary classification of network resources.

- Like Harvest, Stabilis adopts the data-conversion-and-migration approach rather than the query-translation-and-decomposition approach based on gateways (or filters) between information systems — a gateway may be a bottleneck and a source of communications delay, thereby compromising scalability.

Other systems that use network resource names and attributes as the basis for search are the descriptive naming systems or attribute-based naming systems. Their main motivation is that a naming tree cannot be searched with flat search requests, but rather must be traversed. Examples include Profile [17], Univers [5], Semantic File Systems [12] and Prospero [15]. Other systems, such as X.500 [21], and the work described in [19], try to blend the hierarchical and the attribute-based naming models. Nebula [4], in particular, is a file system that permits files to be organised (contained and scoped) according to different views. Similarly, Stabilis provides views to differently organise classes of objects and scope queries.

**Object-oriented databases:** Stabilis metainformation describes schemas in the same way that a data dictionary describes logical schemas in databases. Also, Stabilis makes use of indices to maintain references to information objects and provides a query language in the same way that most object-oriented databases do. Some object-oriented query languages seek compatibility with SQL. For this reason, they provide relational operations, such as join and projection. This permits query results to be new objects of new classes. In contrast, Stabilis query language ensures that query results are always entire and existent objects of already defined classes (schema conservative). Indeed, the join operation is equivalent to navigation using path expressions, i.e., associative access, as explained in [2]. Our approach to query language is its integration with an existent programming language and corresponding type system, as it is the case of Ontos [1] with C++, and GemStone [10] with Smalltalk. In fact, this feature of Stabilis permitted it to be integrated with a system aimed at the management of programs [6].
7 Concluding Remarks

We have developed a toolkit that demonstrates the feasibility of distributed object-oriented metainformation systems (i.e., brokers that conform to the object-oriented approach to information modelling, running on distributed computing environments) and that consolidates concepts found in information discovery tools, object-oriented databases and distributed systems. Our preliminary experiments with the toolkit indicate that Stabilis brokers are an effective means of manipulating information objects, due both to the power of object-oriented modelling and high availability and consistency provided by the underlying distributed transaction facility. The representation of schemas as metaobjects permits users to discover what information a Stabilis broker indexes, and eases the automatic generation of programs. The toolkit is highly portable since it is implemented using only a standard programming language and operating system. Some implementation issues remain to be tackled to enable the support of more realistic information bases, i.e., very large bases that span wide-area networks. This includes a more sophisticated data structure for indices, such as B-trees, and the support of method invocation in queries. Some other areas of application we are currently investigating include: travel agencies, computer network resources, office documents and department stores. A straightforward extension to our work would be to make Stabilis brokers accessible via the World Wide Web; this work is also currently under way.

Acknowledgments

Our thanks are due to all the members of the Arjuna team as they have been very supportive throughout the development of Stabilis, particularly Graham D. Parrington, Stuart M. Wheater, Mark C. Little and Steve J. Caughey. Special thanks go to Luiz E. Buzato, who was our partner in the development of most of Stabilis features and has now returned to his previous position with the University of Campinas, Brazil.
Appendix A: Stabilis Query Grammar

The algebraic query language syntax is defined by the context-free grammar below. The grammar is specified by listing their productions. Each production defines a non-terminal symbol, called the left side of the production, through an Extended Backus-Naur Form (EBNF) expression, called the right side of the production. The start symbol of the syntax is the non-terminal defined by the first production, that is, the symbol query. The notation used is shown in the following Table:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>Separation between left and right side</td>
</tr>
<tr>
<td>$\bullet$</td>
<td>Termination of a production</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$\lbrace x \rbrace$</td>
<td>A sequence of zero or more instances of $x$</td>
</tr>
<tr>
<td>$Not(x)$</td>
<td>Set complement in relation to $x$ in a regular expression</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Regular expression denoting the empty string</td>
</tr>
<tr>
<td>$'xyz'$</td>
<td>The terminal symbol $xyz$</td>
</tr>
<tr>
<td>$xyz$</td>
<td>The non-terminal symbol $xyz$</td>
</tr>
</tbody>
</table>

query $\rightarrow \epsilon \mid$ expression $\bullet$

epression $\rightarrow$ class_expression $|\text{intersection_expression}|$
\hspace{1cm}union_expression $| (' expression ')' $\bullet$

class_expression $\rightarrow$ class_identifier where_clause $\bullet$

intersection_expression $\rightarrow$ expression $'k'$ expression $\bullet$

union_expression $\rightarrow$ expression $'l'$ expression $\bullet$

where_clause $\rightarrow$ $('(' where_expression ')')$ $\bullet$

where_expression $\rightarrow$ $\epsilon \mid$ attribute_expression $\bullet$

attribute_expression $\rightarrow$ term $|\text{and_expression}|$
\hspace{1cm}or_expression $| '(' attribute_expression ')')$ $\bullet$

term $\rightarrow$ attribute_term $|\text{role_term}$ $\bullet$

and_expression $\rightarrow$ attribute_expression $'&$' attribute_expression $\bullet$

or_expression $\rightarrow$ attribute_expression $'|$ attribute_expression $\bullet$
References
