SPEAR: Scalable Pub-Sub Architectures for Efficiency and Robustness
Design Document

Sagar Chaki

August 30, 2002

Contents

Chapter 1

Introduction

This is the design document for the Scalable Pub-sub architectures for Efficiency And Robustness (SPEAR) project. The aim of the project is to develop paradigms and techniques for advancing the state-of-the-art in pub-sub technology. The benefits of these breakthroughs will be made available to the end-user via a set of tools. This document describes the design, implementation and usage of these tools.

Any pub-sub systems consists primarily of two types of components viz. the clients and the servers. The clients are further classified into publishers and subscribers depending on whether they are providing publications or subscriptions to the servers. The servers, often called brokers, are the workhorses of the system. They are responsible for accepting publications and subscriptions from the clients, matching publications to subscriptions, and notifying appropriate subscribers. Accordingly, the architecture of SPEAR can be broadly partitioned into the client and the server side. The server provided by SPEAR is called BLADE and the client is called SHAFT. Although a pub-sub system might consist of multiple instances of BLADE, these instances will behave as independent standalone units. SPEAR does not deal with issues related to broker-collaboration.

1.1 Performance criteria

To be effective, a broker must satisfy the following performance criteria:

- *Correctness.* It must correctly match publications to subscriptions.
- *Expressivity.* It should be capable of handling a rich set of publications, subscriptions and matching semantics.
- *Scalability.* It should be capable of handling a large number of subscriptions, on the order of 100,000.

- *Efficiency.* It should be capable of matching publications to subscriptions very quickly; something on the order of 1000 publications a second is typically expected.

- *Collaboration.* Multiple brokers should be able to collaborate between themselves to enhance overall performance.

The principal thrust of SPEAR is towards scalability and efficiency. We believe that these issues are of tremendous importance even if all the other features are in place. BLADE is capable of matching 1000 publications against 100,000 subscriptions on a modern high-end server.

Chapter 2

Goals and Tradeoffs

The scalability and efficiency requirements drastically influence the design of SPEAR. The following are the goals for SPEAR.

2.1 Goals

The broker should be capable of accepting new subscriptions. It should also be able to accept publications and match them against existing subscriptions. The primary goal is to design a broker that will be capable of supporting a large number of subscriptions and high publication rate. An acceptable target is to match 1000 publications every second against 100,000 subscriptions on a 1 GHz machine with about 512 MB of RAM. The broker should also allow the deletion of existing subscriptions. Thus there are three kinds of client requests - subscriptions, unsubscriptions and publications. However, subscriptions and unsubscriptions can be assumed to occur very rarely.

The broker should also be as generic as possible. This means that it should export a very basic set of functionalities. However the API should be powerful enough so that complicated semantics can be easily be built on top of it. For example, the broker should be capable of using different kinds of matching semantics for different kinds of subscriptions and publications. It should be fairly easy to start up a broker and use its services. Ease of porting to various OS and hardware platforms is also a desirable feature. We now describe some tradeoffs that were made owing to these design considerations.

2.2 Tradeoffs

The main tradeoffs are with respect to the choice of the implementation language, the structure of subscriptions and publications, the matching semantics, the communication mechanism, the concurrency alternatives and real-time response.

2.2.1 Choice of implementation language

The choice of the implementation language is very crucial to the development of any software. We chose C++ as the implementation language for BLADE. First, C++ is extremely efficient. Moreover, the implementation of BLADE requires the use of Binary Decision Diagram (BDD) libraries. There exists an efficient BDD library written in C (but which can be easily ported to C++). Furthermore, if BLADE were written in Java, software written in C++ would not be able to use it in a natural manner. On the other hand, owing to the native method invocation feature, applications written in Java would be easily able to use any implementation of BLADE in C++. C++ was selected as the implementation language owing to its O-O features and the availability of STL classes that are used quite heavily.

The above restrictions do not apply to SHAFT. Although the current implementation of SHAFT also uses C++, we could easily envision implementations in other popular languages like Java.

2.2.2 Structure of subscriptions and publications

The broker is capable of accepting subscriptions that are atomic queries on message attributes and boolean combinations thereof. It is also capable of accepting publications that are (attribute, value) pairs. While this may appear to be overly restrictive, we believe that in reality it will turn out to be otherwise. It should be fairly easy to translate more complicated subscriptions (like XQL queries) and publications (like XML documents) to this simple format. We expect that the simplicity of the broker's input language will go a long way in enhancing its generality.

2.2.3 Matching semantics

Another issue that needs to be addressed is the choice of matching semantics. Different subscriptions could be associated with different semantics. We have identified three distinct semantics that appear interesting:

- SiSL - Simple Subscription Language
- DeSL - Default Subscription Language
- StSL - Strict Subscription Language

Of these, the SiSL is only applicable to complete messages. In fact both DeSL and StSL are equivalent to SiSL for complete messages. Hence the broker really needs to implement DeSL and StSL. There are broadly two possibilities. One choice is that each instance of the broker could handle both these semantics. In other words, a single broker could handle subscriptions associated with any semantics. The other choice is to associate each instance of the broker with a particular semantics. The broker instance is only responsible for subscriptions associated with that particular semantics. It is the duty of some higher-level program to direct only appropriate subscriptions to it. We have taken the second option for ease of implementation. Since a program can create multiple broker instances and since the matching processes for different semantics are independent, we believe our choice will not restrict the effectiveness of the system.

2.2.4 Communication

There is also the need to set up a notification mechanism by means of which the broker can notify the appropriate client when a message gets successfully matched against a subscription. This mechanism could be constructed in various ways. One approach is to use different means of communication depending on the locations of the broker and the client. For example, if the two processes reside on the same host, we could use some fast IPC mechanism like *shared memory*, *message queue* or *pipes*. However for communications across multiple hosts we would resort to comparatively slower TCP/IP sockets.

Our approach is to use TCP/IP sockets in all cases. The advantage is again ease of implementation and uniformity. The potential disadvantage is loss of performance since we would be using a slower means of communication even if the client were on the same host as the broker. However we believe that this would not be a serious issue for two reasons. First, the situation where a client and the broker are on the same host is expected to be very unlikely in the domains, like *e-commerce*, which are of interested to us. Second, in most modern operating systems, network packets destined to the local host go directly to the appropriate port instead of via the regular protocol stack. Thus socket communication between processes on the same host is expected to be comparable in speed to communication via IPC mechanisms.

2.2.5 Alternative BDD libraries

The implementation of BLADE requires the use of a BDD library. The Colorado University BDD package (CUDD) version 2.3.0 was used due its wide popularity, stability, performance and ease of use. Some other BDD packages are those by David Long and BUDDY.

2.2.6 Concurrency alternatives

There Are Typically Two Types Of Concurrent Servers, Those That Use A New Thread For Every Client Connection And Those That Use A Fixed Pool Of Threads To Serve All Client Requests. The Servers That Use A Fixed Pool Of Threads Come In Two Flavors. Some Servers Respond To The Entire Request Of A Client Process By A Single Thread. In Other Servers, Every Request Is Processed In Several Stages, Each Being Performed By A Separate Thread. Blade's Design Can Be Best Described As A Hybrid Of All These Concepts. It Consists Of Three Distinct Stages. Each Stage Communicates With The Next Through A Set Of Shared Data-Structures. These Stages Correspond

Naturally To The Steps Involved In The Servicing Of The Different Kinds Of Client Requests.

The First Stage Is Responsible For Accepting Client Requests And Uses A New Thread For Every Client Connection. The Remaining Two Stages Use A Fixed Number Of Threads And Actually Service These Requests. It Is Debatable Whether Or Not The Front-End Could Use A Fixed Pool Of Processes To Accept The Client Requests. It Seems That This Would Be More Appropriate If Each Client Connection Was Of Very Short Duration Since. If A New Thread Were Spawned For Every New Client Connection In Such A Situation, The Overhead Of Creating And Destroying Threads Would Be Quite High. However The Current Implementation Of Blade Allows A Client To Make An Unlimited Number Of Requests Per Connection. Hence Using A New Thread Per Connection Seems Sensible. In Situations Where The Connections Are Expected To Be Short, A Fixed Pool Of Threads Might Be More Desirable.

2.2.7 Real-time response

Since the addition and deletion of subscriptions are extremely expensive operations, they are performed very infrequently. Thus there is no guarantee that a particular subscription (or unsubscription) request will take effect as soon as it is accepted. However the frequency with which these requests are services is user configurable. In situations where real-time response is crucial, the broker can be configured to service requests as soon as it receives them. This will result in requests taking effect in as close to real-time as possible.

The implementation of BLADE allows simultaneous update of the set of subscriptions and matching of publications. The matching does not need to be halted while the set of subscriptions is being updated.

Chapter 3

Software Architecture

SPEAR provides the application developer with a set of libraries. The API consists of two classes, called Blade and Shaft, which stand for the broker and the client respectively. In order to use them, the application has to include the supplied header files and link the libraries during compilation. The classes export a set of public methods that can be used to invoke their services. A program might create objects of both classes and as many objects of each class as it wishes. However all these objects will have no reciprocal knowledge. In other words, they will behave as independent units. The program must do any coordination between them explicitly. The classes and the exported methods are described in greater detail later. We now describe the overall architecture of SPEAR.

3.1 Overarching Architectural Paradigm

The overall design paradigm followed in SPEAR is the client-server paradigm. The server is multithreaded and multistaged and manages simultaneous access to data structures by different stages that facilitate additional and removal of subscriptions and matching of publications. Figure 3.1 summarizes the key features of SPEAR's design. The rectangles indicate functional blocks and data structures while the arrows indicate the flow of data.

3.2 System Decomposition

As figure 1 indicates, SPEAR can be broadly decomposed into the client and the broker components. We now describe each of these in greater details.

3.2.1 Client Decomposition

The architecture of the client is quite simple. It is simply responsible for shipping off requests to the broker. It does so by opening a communication channel with

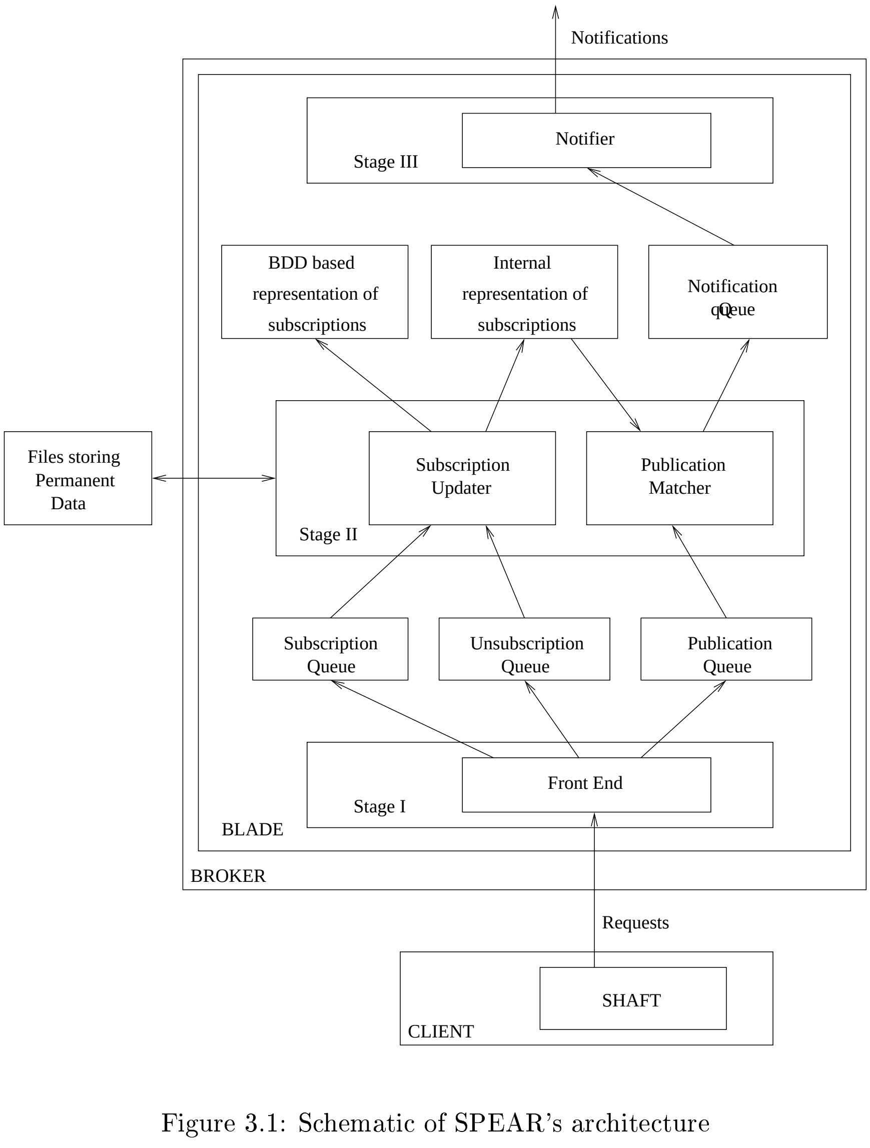

Figure 3.1: Schematic of SPEAR's architecture

the server (a socket connection) and then communicating with the server using a well-defined protocol. There is no limit to the number of requests it can make per connection. The client is exported via the Shaft class of the library. The services of the client can be invoked via the pubic methods of this class. These methods as well as the protocol between the client and the server will be described in greater detail later.

3.2.2 Server Decomposition

The architecture of the server is more complex. As can be seen from figure 1, there are three major stages in the structure of BLADE.

Stage I

Stage I can be considered as the server front-end. It is responsible for accepting client requests over the network. At the very outset it opens a socket connection on a predefined port. At any point of time there is always one thread, called the main thread that listens to the socket. Once a client connection is accepted, the main thread spawns a new thread and hands over the responsibility of listening to the socket to the newly created thread. In other words, the spawned thread becomes the new main thread. The old main thread itself services the requests of the new client. Once this client connection has been terminated, the old main thread exits. Servicing a client request simply means accepting the request and inserting it in the appropriate queue. Each item inserted in a queue is also assigned a unique id. The server returns the id assigned to the request to the client. This id can be used by any client to refer to a particular request at a later time. For example, in order to request the removal of a subscription, the client just sends the id assigned to the subscription.

Request Queues

These queues form the bridge between stages I and II of the server. Stage I can be viewed as the producer and stage II the consumer for these queues. There are three queues, one for each kind of request. The queues are thread-safe since they must support simultaneous access by multiple threads.

Stage II

Stage II of BLADE is where most of the actual work gets done. It consists of two threads called the Subscription Updater (SU) and the Publication Matcher (PM). As the name implies, these two threads are responsible for updating the set of subscriptions and matching publications to existing subscriptions respectively. Both of these threads iterate over infinite loops.

Subscription Updater During every iteration, the SU processes the pending subscription and unsubscription requests. Since these are expensive operations, they are done infrequently. This is achieved in two ways. First the SU sleeps for a predefined amount of time after each iteration. Second the SU ignores the pending requests until their numbers exceed a predefined threshold. Once a sufficiently large number of requests have piled up, the SU updates the set of subscriptions. These thresholds can be configured to make subscriptions as immediate as required. The SU updates the subscriptions in the following manner:

- It reads in the current subscriptions from the predefined file in permanent storage.
- It reads in the pending subscriptions from the subscription queue and adds them to the list of subscriptions read from the file.
- It reads in the unsubscription requests from the unsubscription queue.
- It removes all subscriptions that were unsubscribed to.
- It stores the updated list of subscriptions permanently to the file.
- It creates the BDD corresponding to these subscriptions.
- It creates the internal representation of the subscriptions from the BDDs.
- It atomically updates the current internal representation. This stage is possible because the PM accesses the internal representation through a pointer. All the SU does is update the value of this pointer to the location of the new internal representation. Note that this means that both the SU and PM can function simultaneously.
- Finally it does cleanup.

Publication Matcher The publication matcher is responsible for matching publications against existing subscriptions. In contrast to the SU, the publication matcher tries to match publications as quickly as possible. In order to enhance performance, the PM could actually match a group of publications together during a particular iteration. The PM performs the following actions during every iteration:

- Extracts a group of publications from the publication queue.
- Matches these publications against existing subscriptions. This is done using the current internal representation of the subscriptions.
- Inserts the resulting notifications into the notification queue.

Notification Queue

The notification queue acts as a bridge between stages II and III of BLADE. It contains all pending notifications that have resulted from matching publications. These notifications are produced by stage II and consumed by stage III.

Stage III

The final stage of BLADE, denoted in figure 1 as the *Notifier*, actually consists of a *main notifier* thread and a pool of *worker notifier threads*. Each worker thread has an individual queue from which it extracts notification requests and processes them. The main thread extracts notifications from the notification queue and distributes them evenly among the queues of the worker threads. Since the number of notification requests generated per second is expected to be very large (potentially much larger than the number of publications), using one thread for each request is not advisable. However some amount of concurrency is advisable for reasons of performance. Using a thread pool seems to fit the bill nicely.

Chapter 4

Data Management

BLADE manages data both in the form of internal data structures and files in permanent storage.

4.1 Data structures

The principal data structures used by BLADE have already been mentioned in the previous section. These data structures provide clean architectural decomposition and interfaces between the various components. They are mentioned again below:

- The publication queue
- The subscription queue
- The unsubscription queue
- The BDD-based representation of the subscriptions
- The internal representation of the subscriptions
- The notification queue

Recall that all the queues used are thread-safe, supporting simultaneous access by multiple threads.

4.2 Persistant Storage

BLADE uses persistent secondary storage to achieve a certain amount of fault-tolerance and consistency. It stores the set of current subscriptions in a file. The name of this file can be passed as a parameter to BLADE before startup. Suppose it is *curr_subs.spear*. An *alternating file protocol* is used to ensure that a consistent set of subscriptions can be recovered even after an abnormal termination of the broker. This protocol uses two versions of the subscription file viz. *curr_subs.spear-1* and *curr_subs.spear-2*.

On startup, and at the beginning of each predicate update cycle, BLADE checks which version of the subscription file is present. If only a single version is present, it loads the subscriptions from that file. If both versions are present, it deletes the version with later modification time and loads the subscriptions from the other version.

At the end of the subscription update cycle, after the set of subscriptions have been update internally, BLADE writes out the updated set of subscriptions to the version of the file that is absent. Note that at this time at least one version of the subscription file must be absent. The only way this condition could be violated is if some other program is creating and/or removing these subscription files. This is illegal and if BLADE detects such a condition, it exits.

The name of the protocol is inspired by the fact that under normal execution conditions, successive sets of current subscriptions are written to alternate versions of the subscription file. Thus BLADE provides persistence of the subscriptions across multiple executions of the broker even under abnormal termination conditions. Note that no such persistence is provided for the publications.

Chapter 5

Concurrency Identification

The implementation of BLADE is multithreaded. The various threads that participate in its functioning have been already described in the previous section.

Chapter 6

Hardware/Software

SPEAR has been implemented and tested on x86-based systems running RedHat Linux 5.2, 6.2 and 7.1. The CUDD package (version 2.3.1) was used for performing various BDD operations. The SPEAR-library was written using ANSI C++ and compiled using g++ (version 2.96-85). The POSIX *pthread* library was used for thread management. The thread-safe queue and a class for doing efficient bit operations were obtained in source format from the web. Since everything is available in source form and is written in ANSI C/C++, we do not expect much difficulty in porting SPEAR to other platforms. In fact a port of the library for Solaris is already available.

Chapter 7

Global Resource Handling

Chapter 8

Communication Protocol

This section describes in detail the communication protocol used by a client and a broker. In the following sections, all terms composed of capitals and underscores are enumeration constants defined in the software, and are used to explicate the protocol. Also, whenever the broker or the client fails in sending some data to the other party, it assumes that the connection has been broken and aborts. We first describe the initial handshaking.

8.1 Handshaking

- Client sends the version number of the protocol it is using.
- Broker receives the version number and checks if it matches its own version number. If so it sends SPBR_VERSION_OK to the client and continues the connection. Otherwise it sends SPBR_VERSION_NOK and terminates the connection.
- Client receives the broker's response. If the response is SPBR_VERSION_OK, it continues with the connection; otherwise it terminates the connection.

We now describe how the client submits different requests and how the broker responds to them.

8.2 Requests

- Client sends a request type to the broker. The valid request types are:
  - SPBR_REQ_PUB_ACK - publication with acknowledgment
  - SPBR_REQ_PUB_NAK - publication without acknowledgment
  - SPBR_REQ_SUB_ACK - subscription with acknowledgment

- SPBR_REQ_SUB_NAK - subscription without acknowledgment
- SPBR_REQ_UNSUB_ACK - unsubscription with acknowledgment
- SPBR_REQ_UNSUB_NAK - unsubscription without acknowledgment
- SPBR_REQ_CONNECT - initiate connection for notification
- SPBR_REQ_QUERY - query the status of a notification connection
- SPBR_REQ_CLOSE - close a notification connection
- SPBR_REQ_QUIT - connection termination request

- Broker receives the request type and checks its validity. If it is valid (and is not of type SPBR_REQ_QUIT), it returns SPBR_REQ_OK and continues processing the request; otherwise it returns SPBR_REQ_NOK and aborts processing the request.

- Client receives the broker's response. If the response is SPBR_REQ_OK, it continues sending the request; otherwise it aborts sending the request.

At this point, if both parties have decided to go ahead, different things happen depending on the type of request being submitted by the client. We describe each separately. Notice that publication, subscription and unsubscription each have two forms of requests. The first kind of request causes broker to acknowledge that the request has been inserted in the appropriate queue for processing by the backend. The second kind of request causes the broker to not send any such acknowledgment. Which of these requests will be made depends on what level of assurance the client from the broker.

8.2.1 Publication Request

- Client sends the length of the publication string followed by the actual string.

- Broker receives the length and then the actual publication. If there is an error in receiving, it aborts processing the request. Otherwise, it tries to parse the publication. If parsing succeeds, it sends SPBR_PUB_OK to the client and continues processing the request; else it sends SPBR_PUB_NOK to the client and aborts processing the request.

- Client receives the broker's response. If the response is not SPBR_PUB_OK, it aborts. Otherwise it sends the following in order:

  - Length of the id string of the publication.
  - Id string of the publication.

  This information is to be used by the broker for notification if the publication is successfully matched against some subscription.

- Broker receives the information sent by the client in order. It sends SPBR_PUB_INFO_OK to the client and continues. If there was a problem in receiving the information from the client, it aborts.
- Client receives broker's response. It it is SPBR_PUB_INFO_OK, it continues with the request; else it aborts.
- If an acknowledgment was not requested then the client completes the request gracefully at this point.
- Broker inserts the publication in the publication queue and if an acknowledgment was requested it sends SPBR_PUB_INQ to the client.
- Client receives broker's response. If it is SPBR_PUB_INQ, it completes the request properly; otherwise it aborts.

8.2.2 Subscription Request

- Client sends the length of the subscription string followed by the actual string.
- Broker receives the length and then the actual subscription. If there is an error in receiving, it aborts processing the request. Otherwise, it tries to parse the subscription. If parsing succeeds, it sends SPBR_SUB_OK to the client and continues processing the request, else it sends SPBR_SUB_NOK to the client and aborts processing the request.
- Client receives the broker's response. If the response is not SPBR_SUB_OK, it aborts. Otherwise it sends the following in order:
  - Length of the id string of the subscription.
  - Id string of the subscription.
  - Length of an IP address string.
  - IP address string.
  - Port number.
  - Validity duration of the subscription.

  This id string, the IP address and the port number are to be used by the broker for notification if some publication is successfully matched against the subscription. The id string is also to be used to identify the subscription whose removal is being requested during an unsubscription.

  The validity duration is to be used to determine if the subscription has expired. No notification is sent if an expired subscription is matched by a publication and all expired subscriptions are automatically removed during the periodic subscription update phase.

- Broker receives the information sent by the client in order. It sends SPBR_SUB_INFO_OK to the client and continues. If there was a problem in receiving the information from the client, it aborts.
- Client receives broker's response. It it is SPBR_SUB_INFO_OK, it continues with the request; else it aborts.
- If an acknowledgment was not requested then the client completes the request gracefully at this point.
- Broker inserts the subscription in the subscription queue and if an acknowledgment was requested it sends SPBR_SUB_INQ to the client.
- Client receives broker's response. If it is SPBR_SUB_INQ, it completes the request properly; otherwise it aborts.

8.2.3 Unsubscription Request

- Client sends the length of the id string followed by the id string of the subscription to be removed to the broker.
- Broker receives the information sent by the client and sends back SPBR_UNSUB_OK to the client.
- Client receives the broker's response. If the response is SPBR_UNSUB_OK, it continues with the request; otherwise it aborts.
- If an acknowledgment was not requested then the client completes the request gracefully at this point.
- Broker inserts the unsubscription in the unsubscription queue and if an acknowledgment was requested, it sends SPBR_UNSUB_INQ to the client.
- Client receives broker's response. If it is SPBR_UNSUB_INQ, it completes the request properly; otherwise it aborts.

8.2.4 Connection Request

- Client sends the length of the IP address string followed by the actual string, followed by the port number, followed by the amount of delay before the connection should be attempted. The IP address and port refer to the server socket with which the broker is expected to establish a connection.
- Broker receives the information sent by the client in order and responds with SPBR_CONNECT_INFO_OK. If there was a problem in receiving the information from the client, it aborts.
- Client receives broker's response. It it is SPBR_CONNECT_INFO_OK, it continues with the request; else it aborts.

- Broker checks if the specified connection already exists. If that is the case, it returns SPBR_CONNECT_EXISTS to the client. Otherwise it waits for the specified amount of time and then tries to establish a socket connection at the specified IP address and port and do the handshake. If the connection was established and handshake completed successfully it responds with SPBR_CONNECT_OK; else it responds with SPBR_CONNECT_NOK.

- Client receives the broker's response. If it is SPBR_CONNECT_OK or SPBR_CONNECT_EXISTS it completes the request gracefully; otherwise it aborts.

8.2.5 Query Request

- Client sends the length of the IP address string followed by the actual string, followed by the port number. The IP address and port refer to the notification connection being queried.

- Broker receives the information sent by the client in order and responds with SPBR_QUERY_INFO_OK. If there was a problem in receiving the information from the client, it aborts.

- Client receives broker's response. It it is SPBR_QUERY_INFO_OK, it continues with the request; else it aborts.

- Broker checks if the specified connection already exists. If that is the case, it returns SPBR_CONNECT_EXISTS to the client. Otherwise it returns SPBR_CONNECT_NOT_EXISTS.

- Client receives the broker's response. If it is SPBR_CONNECT_EXISTS or SPBR_CONNECT_NOT_EXISTS it completes the request gracefully; otherwise it aborts.

8.2.6 Close Request

- Client sends the length of the IP address string followed by the actual string, followed by the port number. The IP address and port refer to the notification connection being closed.

- Broker receives the information sent by the client in order and responds with SPBR_CLOSE_INFO_OK. If there was a problem in receiving the information from the client, it aborts.

- Client receives broker's response. It it is SPBR_CLOSE_INFO_OK, it continues with the request; else it aborts.

- Broker tries to close the specified connection. If it succeeds, it returns SPBR_CLOSE_OK to the client. Otherwise it returns SPBR_CLOSE_NOK.

- Client receives the broker's response. If it is SPBR_CLOSE_OK or SPBR_CLOSE_NOK it completes the request gracefully; otherwise it aborts.

8.2.7 Termination Request

- Both parties terminate the socket connection.

8.3 Notifications

In the case of notifications, the roles of the broker and client are reversed. Further a notification in sent and received exactly the same way as a publication. This is because notifications sent by a broker could actually serve as publications to other brokers. This is necessary to build up a hierarchy of broker's and subscriptions in order to match complex subscriptions.

It should be noted that BLADE does not actively manage any notification connections. It only processes requests sent by clients. Thus all notification connections must be opened and closed explicitly by clients via appropriate requests. For example suppose client C_1 requests the broker to open a notification connection with another client C_2. If C_2 terminates this connection from its side (or if it terminates for some other reason), the broker will not reciprocate by closing the connection as well. Instead it has to be explcitly requested to do so. Only then can this notification connection to C_2 be reestablished.

Chapter 9

Software Interface Design

Since SPEAR provides the application developer with a set of libraries, the only interface available is via the API provided.

9.1 BLADE API

The following is the API of the Blade class.

```
class Blade
{
  public:
    Blade(BladeParams &params);
    ~Blade();
    void Start();
};
```

The argument to the constructor can be used to supply values to various parameters of the broker (like the port number on which the server socket must be opened etc.) All parameters have default values that are used in the absence of values supplied by the user.

The following program first creates a broker with DeSL semantics, appropriate default values and *curr_subs.spear* as the subscription file. It then starts up the broker at port 9999 with a backlog of 15 (the backlog is the maximum number of connection requests that can queue up on the broker socket).

```
#include "Blade.h"

void main()
{
  BladeParams params;
  params.SetMatchSem("default");
  params.SetDefaultInt(-10);
```

```
  params.SetDefaultDbl(-2.673);
  params.SetDefaultStr("foo");
  params.SetPort(9999);
  params.SetBacklog(15);

  Blade *blade = new Blade(params);
  blade->Start();
}
```

9.2 SHAFT API

The following is the API of the Shaft class.

```
class Shaft
{
  public:
    Shaft();
    ~Shaft();
    int Connect(const ConnectInfo &connectInfo);
    void Disconnect();
    bool Publish(PubInfo &pubInfo);
    bool Subscribe(SubInfo &subInfo);
    bool Unsubscribe(UnsubInfo &unsubInfo);
    bool Open(const ConnectInfo &connectInfo);
    int StartNotConnect(Shaft &srvShaft,Shaft &newShaft,
        const ConnectInfo &connectInfo);
    int StartNotConnect(const ConnectInfo &connectInfo);
    bool AcceptNotConnect(SpearSocket &srvSock);
    int QueryNotConnect(const ConnectInfo &connectInfo);
    int CloseNotConnect(const ConnectInfo &connectInfo);
    bool GetNotification(NotInfo &notInfo);
};
```

We now describe each of the public methods of *Shaft* in greater detail:

- **Connect**: Opens a socket connection to the broker and does the initial handshaking. The argument of the method contains all necessary data for establishing the connection. Returns 0 on success, -1 if a socket connection could not be established, -2 if there was a protocol version mismatch and -3 if some other error occurred.
- **Disconnect**: Sends the connection termination request and terminates the connection.
- **Publish**: Sends a publication request. Returns *true* on success and *false* otherwise.

- **Subscribe**: Sends a subscription request. Returns *true* on success and *false* otherwise.
- **Unsubscribe**: Sends an unsubscription request. Returns *true* on success and *false* otherwise.
- **Open**: Opens a server socket that can be used to receive notifications from the broker.
- **StartNotConnect**: There are two versions of this method. Both versions try to make the broker open a new notification connection. In the first version, the server socket and the new socket resulting from the open connection are passed as arguments. This is intended to be used when the same thread serves as the receiver of notifications. The second version assumes that the server socket exists and is being watched by another thread or process.
- **AcceptNotConnect**: Accepts a notification connection from a broker. Returns *true* on success and *false* on failure.
- **QueryNotConnect**: Queries about the existence of a notification connection. Returns 0 if the connection exists, -1 if the connection does not exist and -2 if some other error occurred.
- **CloseNotConnect**: Closes a notification connection. Returns 0 if the connection was closed, -1 if the connection was not closed and -2 if some other error occurred.
- **GetNotification**: Receives one notification from the broker. The details of the notification are stored in the argument provided.

Note that any call to *Publish*, *Subscribe* or *Unsubscribe* must be preceded by a call to *Connect*. Assuming that the broker was started on a machine with IP address "100.100.100.100", the following program connects, subscribes, publishes, removes its subscription and finally disconnects.

```
#include "Shaft.h"

void main()
{
  ConnectInfo cInfo;
  cInfo.SetIPAddress("100.100.100.100");
  cInfo.SetPort(9999);

  Shaft shaft;
  shaft.Connect(cInfo);

  SubInfo sInfo;
  sInfo.SetSub("(a > 10) & (a < 20)");
```

```
  shaft.Subscribe(sInfo);

  PubInfo pInfo;
  pInfo.SetPub("a 15");
  shaft.Publish(pInfo);

  UnsubInfo uInfo;
  uInfo.SetSubId(sInfo.GetId());
  shaft.Unsubscribe(uInfo);

  shaft.Disconnect();
}
```

Chapter 10

Conclusion

In this section we mention some future directions in which the SPEAR package could be enhanced.

Currently only the DeSL semantics has been implemented.

One issue is that of attribute types. Currently BLADE allows only three kinds of attributes viz. integer, double and string. It would be desirable to allow additional domain specific attribute types, like money, each with their own operations. In fact it would be desirable to be able to dynamically add new attribute types and associated operations without having to shut the broker down.