

S O F T W I R E D



# Programmer's Manual

Version 0.5

August 10, 1998

SoftWired AG  
Technoparkstrasse 1  
8005 Zürich  
[ibus@softwired.ch](mailto:ibus@softwired.ch)  
[www.softwired.ch](http://www.softwired.ch)



© Copyright 1998 SoftWired AG. All rights reserved.  
Document-reference: Programmers\_Manual.fm

# Table of Contents

<b>1 Why iBus?</b>	<b>7</b>
<b>2 Getting Started</b>	<b>8</b>
2-1 An Example Application	8
2-2 Sending and Receiving Data	10
2-3 Configuring Your Environment	10
<b>3 Uniform Resource Locators</b>	<b>12</b>
3-1 Why Those Addresses?	12
3-2 URL Factory	12
3-3 Demultiplexing	13
3-4 Code Example	13
<b>4 Protocol Stacks</b>	<b>15</b>
4-1 The Main Protocol Objects	16
4-2 Summary of Features	18
4-3 Quality of Service Strings	18
4-4 Aliases	19
4-5 Code Example	19
<b>5 Postings</b>	<b>21</b>
5-1 Code Example	21
<b>6 Pushing and Pulling Postings</b>	<b>23</b>
6-1 Pushing Data	23
6-2 Pulling Data	23
6-3 Opening Transmission Channels	24
6-4 Subscribing Receiver Objects	25
6-5 IP Multicast Communication	25
6-6 TCP Communication	26
6-7 Code Example	28
<b>7 Membership and Failure Notification</b>	<b>30</b>
7-1 Definition of Terms	30
7-2 Membership Classes	31
7-3 Registering a Membership Monitor	32
7-4 Time-Outs	32
7-5 Code Example	33
<b>8 Logging Events</b>	<b>35</b>
8-1 Log Levels	35
8-2 Logging Events in Java Applications	36
8-3 Redirecting Log Output	36
8-4 Code Example	37



<b>9 Threads</b>	<b>38</b>
9-1 Single-Threaded Dispatch	38
9-2 Thread-Per-Request	38
9-3 Thread-Pool	39
9-4 Thread-Per-Channel	39
9-5 Comparison	39
9-6 Code Example	39
<b>A iBus System Properties</b>	<b>41</b>
<b>B A Note about Configuration Management</b>	<b>42</b>

*This document describes the writing of iBus applications. It refers to the iBus `javaDoc` HTML documentation when explaining programming interfaces. An arrow notation ( $\Rightarrow$  See ...) is used to refer to information provided in the HTML documentation.*

*Examples of iBus applications are provided in the `iBus\example` directory which is part of the documentation JAR file. The development of iBus protocol objects is not addressed by this manual.*



# 1 Why iBus?

iBus is a 100% Java™ middleware aimed to support intranet applications such as content delivery systems, groupware, financial news distribution systems, fault-tolerant client-server systems, and multimedia applications. iBus provides multicast channels that allow Java applications to interact by a push/pull/subscribe communication paradigm. The system is conceived to run atop TCP and IP multicast. Besides serving as a versatile communication platform, iBus also performs coordination tasks such as notifying applications when other applications they depend on start or fail.

iBus can be used along with JDK Serialization, AWT, Java Beans, Java IDL, Java Media Framework and other packages.

iBus supports the development of location independent applications that are relocated from one machine to another without affecting their peer applications. The iBus architecture has no single point of failure and there are no background services that need to be present in order to use iBus.

iBus provides a quality of service framework in which applications only pay for services they need: programmers request qualities of service such as reliable and unreliable multicast, reliable and unreliable point-to-point communication, and failure detection. The protocol composition framework that is part of the package allows programmers to extend iBus with yet unsupported qualities of service, for example message compression or encryption.



## 2 Getting Started

iBus is best introduced by providing a simple yet realistic example that uses the push and subscription APIs.

### 2-1 An Example Application

Consider an application that reads foreign exchange data from a satellite feed and transmits ticks<sup>1</sup> to a large number of trader workstations for display in a AWT widget. The application consists of a `TickTalker` and of a `TickListener` tool.

The talker runs on a workstation attached to the satellite feed, it uses the `getNextTick()` method to read the next tick from the satellite feed<sup>2</sup>. The talker then multicasts the tick to all receivers.

A listener application receives those tick objects and passes them along to the `drawTick()` method for display. The talker application consists of class `TickTalker`:

```
import iBus.*;
import FinancialInfo.Tick;
import DataFeeds.SatelliteFeed;
...

// tick talker application.
public class TickTalker {
    public static void main(String argv[])
        throws Exception
    {
        // declare a destination URL, a protocol stack
        // for reliable multicast, and a posting object:
        final iBusURL url = iBusURLFactory.create("StockEx-Feed",
            "1.0", "/ticks");

        final Stack stack = new Stack("Reliable");
        final Posting posting = new Posting();

        // connect to the satellite feed:
        final SatelliteFeed satellite
            = new SatelliteFeed("Reuters:Triarch1");

        // allocate one slot in the posting:
        posting.setLength(1);

        // register this application as a talker on the tick channel:
        stack.registerTalker(url);
```

- 
1. A tick is a unit of financial information, such as the US Dollar / Yen exchange rate at a given point in time.
  2. Here we assume that the satellite software buffers incoming ticks and that `getNextTick()` reads from the buffer.



```

        // in a loop, get the next tick and push it:
        for (;;) {
            // get the next tick from satellite feed:
            final Tick t = satellite.getNextTick();

            // multicast the tick:
            posting.setObject(0, t);
            stack.push(url, posting);
        }
    }
}

```

The listener application consists of class `TickListener` and `TickDispatch`. `iBus` feeds incoming ticks into a `TickDispatch` object by calling its `dispatchPush` method:

```

import iBus.*;
import FinancialInfo.Tick;
import FinancialInfo.GraphWidget;
...

// tick listener application.
public class TickListener {
    public static void main(String argv[])
        throws Exception
    {
        // declare a destination URL and a protocol stack
        // for reliable multicast (must match the talker application):
        final iBusURL url = iBusURLFactory.create("StockEx-Feed",
            "1.0", "/ticks");
        final Stack stack = new Stack("Reliable");

        // create a receiver object that dispatches incoming postings:
        final TickDispatch disp = new TickDispatch();

        // subscribe the receiver object to the tick channel:
        stack.subscribe(url, disp);

        // wait for incoming postings:
        stack.waitTillExit();
    }
}

// This class is needed for receiving Postings on the tick channel.
class TickDispatch implements iBus.Receiver {
    public void dispatchPush(iBusURL source, Posting p)
    {
        // retrieve the tick from the posting:
        Tick t = (Tick)p.getObject(0);

        // draw the tick in a AWT graph widget:
    }
}

```



```
graph_.drawTick(t);  
}  
  
// (error and dispatchPull method not shown here.)  
  
private GraphWidget graph_ = new GraphWidget("Tick Display");  
}
```

## 2-2 Sending and Receiving Data

In the example above ticks are transmitted efficiently by multicast. iBus exploits hardware multicast facilities as available in Ethernet and Tokenring LANs. The network load is thus constant and independent of the number of `TickListeners` that tap into the tick channel. You can relocate talkers and listeners from one machine to another dynamically without need to restart any of their peer applications. iBus enables a model of *spontaneous networking* where applications join and leave channels dynamically.

Neither registries nor naming services need to be contacted by iBus when an application requests to join a channel. iBus provides the abstraction of a *ubiquitous information bus* on behalf of which applications exchange events.

Class `Tick` only needs to implement interface `java.io.Serializable` in order to be transmitted via an iBus channel. A `Tick` object would typically provide a string describing a cross rate, for example `"USD/JPY"`, and an exchange rate, for example `129.900`.

iBus implements a *publication/subscription* paradigm where self-describing data objects, called *postings* in iBus terminology, are injected into communication channels. Communication is typically one-to-many and asynchronous, although point-to-point communication and synchronous invocation are supported as well. Talker applications *push* postings into one or more channels, listener applications *subscribe* to one or more channels to receive postings. In addition a *pull* operation is provided that works much like RMI. Pull is a two-way operation used to explicitly request data from an iBus application.

A quality of service such as reliable multicast or encrypted communication is tied to a channel. In iBus a quality of service is represented by a *protocol stack*. Talkers and listeners need to agree on a URL and on a protocol stack to exchange information.

## 2-3 Configuring Your Environment

In order to build and run iBus applications you need a copy of JDK-1.1.1 or higher. The location of the `iBus.jar` file needs to be included in your `CLASSPATH` environment variable. For example

```
setenv CLASSPATH "${CLASSPATH}:/usr/local/lib/java/ibus.jar"
```

To test your set up enter: `javap iBus.Stack`. This will print the interface of class `Stack` showing that the `iBus` package is accessible via the class path.



## 3 Uniform Resource Locators

⇒ See *iBus.iBusURL*

A *iBusURL* object denotes a communication **channel** by which Java applications exchange *postings*. iBus URLs are of the form

```
ibus://<address>[:<port>]<subject>
```

The **address** is a class A, B, C or D (multicast) IP address or a host name. IP multicast is the predominant communication mechanism in iBus. Point-to-point communication is normally used only within the iBus software to deliver acknowledgements and channel membership management messages. However, programmers are free to subscribe and to post to point-to-point URLs by TCP or UDP. If no **port** number is specified then a default of 8733 is used. If 0 is provided as the port number then iBus will pick the next free port number and assign it to the URL. This is useful when creating server applications that use an iBus TCP stack.

The **subject** is an application specific hierarchical string to fully denote an iBus communication channel. The following are examples of IP multicast URLs:

```
ibus://226.1.1.33/ny-stockex/APL
ibus://226.1.1.33/ny-stockex/SUN
ibus://226.1.1.33:9111/ny-stockex/IBM
ibus://226.1.2.1/inhouse/teleconf/10am
ibus://226.1.2.2/inhouse/financial/news
```

The following URL denotes a point-to-point channel:

```
ibus://myhost.somewhere.com:8111/services/file_server
```

### 3-1 Why Those Addresses?

An early design decision was that iBus applications should not depend on the presence of any iBus registry or name server. Once you have a network set up between two machines you are able to run iBus applications. IP multicast addresses are embedded into iBus URLs such that no name server needs to be present, and to give you maximum flexibility in mapping high-traffic channels to IP multicast addresses. The *iBusURLFactory* class is provided for making the creation of URLs and the assignment of IP multicast addresses easier.

### 3-2 URL Factory

⇒ See *iBus.iBusURLFactory*

Class *iBusURLFactory* helps the application developer in creating URLs that obey a well-defined format. The factory also provides for automatic assignment of IP multicast numbers. For that a hash algorithm is applied to the subject part of the URL to compute a IP multicast address. It is thus guaranteed that a given sub-

ject always maps to the same IP multicast address. We recommend that you always rely on `iBusURLFactory` for generating URLs since this allows you to run a distributed application in multiple **system areas**.

The URL factory constructs URLs of the form:

```
ibus://<address>[:<port>]/<System>/<Service>/<Version>/Subject
```

- `System` denotes the **system area** the application is running in, for example, *production system*, *test system*, or *Joe's development system*. The `System` part can be set only through the `iBusSystem` property as described in Appendix A. It defaults to the name `dev` for development. The remaining three components can be set through the `iBusURLFactory` methods.
- `Service` provides the name of the iBus service the URL is generated for. For example `StockEx-Feed`, `Videoserver`, `Directory411`, etc.
- `Version` provides the version of the application. This allows you to run multiple versions of the same application simultaneously without having the applications interfering with each other.
- `Subject` contains a service specific hierarchical subject, for example `/quotes/sun`, `/movies/forrestgump`, `/inquiries`, etc.

### 3-3 Demultiplexing

The demultiplexing of iBus traffic is done partly in the networking hard- and software, and partly in the iBus software. Postings that are sent to two different IP multicast addresses are typically demultiplexed by the network card. Some Ethernet cards do this in hardware, other leave the demultiplexing to their device driver. Check with your card vendor for details.

Postings pushed to URLs that differ only in their subject part are demultiplexed by iBus. This means that if your application is subscribed to

```
ibus://226.1.2.1/weather/dailyforecast,
```

then postings pushed to

```
ibus://226.1.2.1/movies/forrestgump
```

propagate up to your application's iBus layer where they are discarded. This can place a substantial computational burden on applications. You might thus want to use separate IP addresses for traffic intensive channels to relieve your CPU from doing too much demultiplexing work.

### 3-4 Code Example

The following code fragment demonstrates the creation of URLs and the extraction of information thereof:

```
// using the iBusURL class directly (not recommended):
iBusURL u1 = new iBusURL("ibus://226.1.1.33:9999/ny-stockex/APL");
```



```
// using the iBusURLFactory (recommended):
iBusURL u2 = iBusURLFactory.create("StockEx-Feed",
    "1.0", "/quotes/sun");
iBusURL u3 = iBusURLFactory.create("Videoserver",
    "0.7", "/movies/forrestgump");
iBusURL u4 = iBusURLFactory.create("Directory411",
    "2.1", "/inquiries");

// returns "/ny-stockex/APL"
String subject = u1.getSubject();

// returns 226.1.1.33:
String address = u1.getAddress();

// returns 9999:
int port = u1.getPort();

// returns the iBus default port, typically 8733:
int defPort = iBusURL.getDefaultPort();

// returns "ibus://224.17.251.126:8733/<your name>/
// <StockEx-Feed>/<1.0>/quotes/sun"
String url = u2.toString();

...
```

## 4 Protocol Stacks

⇒ See *iBus.Stack*

Talkers and listeners need to agree on a URL and on a **protocol stack** to be able to exchange data. A protocol stack represents a **quality of service** (QOS) such as reliable multicast, reliable point-to-point streaming, encrypted communication, and so forth. A stack consists of a linear list of **protocol objects**. Postings are always pushed and received on behalf of stacks. Application may use multiple stacks simultaneously.

At the sending side, the posting travels down the stack from the topmost to the bottommost protocol object. The first protocol object might encrypt the posting, the second one fragment large postings into chunks that fit in a network packet, the third one buffer fragments for retransmission, and the bottommost object would typically transmit the fragments by plain IP multicast.

At the receiving side a fragment arrives at the bottommost protocol object in the stack. The next object checks whether any prior fragment is missing. The postings are then assembled and decrypted.

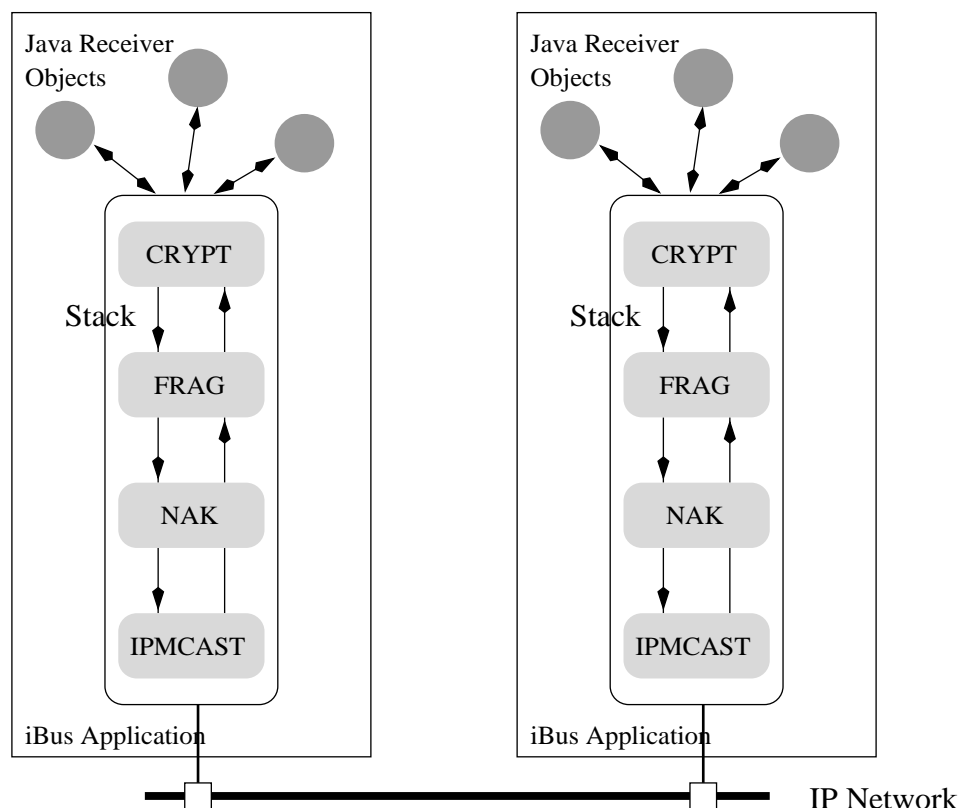


Figure 1: Two iBus applications communicating through protocol stacks



## 4-1 The Main Protocol Objects

⇒ See *iBus.ProtocolObject*, *iBus.layers.IPMCAST*, *iBus.layers.REACH*, etc.

protocol objects are included in the `iBus.layers` package. iBus mainly provides the following protocol objects:

- **IPMCAST** manages `java.net.MulticastSocket` and `java.net.DatagramSocket` objects for sending and receiving UDP datagrams. IPMCAST does not implement any reliability or flow control mechanism. This object can be used stand-alone or in conjunction with REACH and NAK.
- **REACH** is a simple failure detection and membership layer. For each channel the stack is subscribed to as a listener or as a talker, REACH periodically submits a heartbeat message on an internal control channel.

When a remote REACH layer receives a heartbeat for the first time it forms a new `View` object and passes it up the stack. The `View` object includes the URL of the sender of the heartbeat. Section 7 explains how to obtain channel membership notifications from iBus.

If a REACH layer does not receive any heartbeat from a sender for a certain amount of time, it forms a new `View` object that excludes the sender and passes it up the stack. Hence applications will temporarily disagree on the view for a given channel but eventually converge to a consistent view. Hence the name "REACH": it means the layer provides reachability information. REACH is conceived to reside atop a multicast protocol object such as IPMCAST.

- **NAK** is a fully reliable, negative acknowledgements (NAK) multicast layer. NAK uses immediate, receiver-initiated, NAK-based, unicast loss notification combined with originator based unicast and multicast retransmission (selectable). The NAK protocol is similar to RAMP (RFC 1485).

NAK does not implement any FIFO ordering nor the elimination of duplicated messages. This is accomplished by a FIFO layer atop of NAK. NAK only makes sure that each message is received at least once, without caring about duplicates and messages arriving out of FIFO order.

NAK does not implement any failure detection as this is handled by the membership layer beneath NAK (REACH, for example). NAK is fully reliable, i.e., it is both sender and receiver reliable. For that it takes advantage of the membership information which is delivered to it by the membership layer beneath. This increases the reliability of the NAK protocol considerably.

- **FIFO** implements a first-in-first-out ordering of messages as well as duplicate elimination. FIFO receives a stream of messages from beneath. It eliminates any duplicates and orders messages according to their sequence numbers. FIFO does not implement any loss recovery as this is handled by the layers below. FIFO is to be placed atop a reliable multicast object such as NAK.



- **FRAG** fragments and reassembles messages that are larger than a certain threshold size. The fragment size can be specified as part of the QOS string. The default size matches a UDP datagram. FRAG is to be placed atop FIFO.
- **LOCALBUS** is a protocol object for local communication within one virtual machine. Its purpose is to accommodate Java Applets that want to exchange events within one virtual machine or Web browser. This allows one to use iBus much like the InfoBus<sup>TM</sup> product. LOCALBUS does not send or receive any data through a network. LOCALBUS is typically the sole protocol object of its stack.
- **PULL** implements request/reply style communication similar to RMI. Unlike RMI, PULL multicasts requests to all receivers on the channel. Every receiver may return a reply to the initiator of the pull. PULL is to reside on top of a reliable multicast protocol object such as NAK.
- **TCP** is intended for push and pull communication via TCP/IP. When `subscribe` is called on a TCP stack a `java.net.ServerSocket` is created. When `registerTalker` is called a TCP connection is established to the destination address given by the URL. This default behavior can be changed by setting the `talkerconnect` and `listenerconnect` properties as described in Section 6-6.

TCP supports communication between one talker and one listener, between multiple talkers and one listener, and between multiple listeners and one talker.

TCP can be used stand-alone or beneath DISPATCH. None of the PULL, FIFO, FRAG, REACH, and NAK objects are needed by TCP.

- **DISPATCH**. In iBus there is one thread which transports events up the stack and invokes the user defined dispatch methods on `iBus.Receiver` objects. DISPATCH can be used as the topmost protocol object to realize threading policies such as *single dispatch thread* (the default), *thread per request*, *thread pool*, and *thread per channel*.
- **BADNET** is a debugging aid that maliciously duplicates, reorders, and throws away messages.
- **SEQCHK** is a debugging aid to check sequence numbers. It aborts the application when a message is received out of sequence.
- **TEMPLATE** is provided in source code form to be used as a template when developing new protocol objects.



## 4-2 Summary of Features

Protocol Object	Resides atop	Implements Pull Operation	Failure Detection	Buffers incoming messages	Buffers outgoing messages
IPMCAST	(stand-alone)			x	
REACH	IPMCAST		x		
NAK	REACH				x
FIFO	NAK			x	
FRAG	FIFO			x	
LOCALBUS	(stand-alone)				
PULL	NAK	x			
TCP	(stand-alone)	x			
DISPATCH	TCP, NAK, LOCALBUS, ...			x	

Table 1: Summary of protocol object features

## 4-3 Quality of Service Strings

⇒ See *iBus.Stack*

iBus provides the `Stack` helper class for creating a protocol stack out of a stringified QOS representation. The QOS strings you will typically use are listed below.

- "IPMCAST" for plain IP multicast with no flow control and no retransmission of lost postings. This is usually adequate for transmitting audio or video packets in a best-effort manner. Here the maximum size of a posting is limited to 8 kb<sup>3</sup>.
- "REACH:IPMCAST" to add failure detection to the aforementioned quality of service. Here your application will be notified when another application joins the channel, leaves the channel, or crashes. In Section 7 you will learn how to obtain such failure notifications from iBus.
- "DISPATCH:PULL:FRAG:FIFO:NAK:REACH:IPMCAST" for reliable, first-in-first-out ordered multicast. This QOS provides retransmission of lost fragments, delivery of postings in the same order as they were sent, and fragmentation/reassembly of large postings.
- "DISPATCH:TCP" for point-to-point communication via TCP/IP.
- "LOCALBUS" for applications that use iBus to exchange events only locally in the context of one Java virtual machine.

---

3. which is the maximum size of a UDP datagram.

Most protocol objects can be parametrized through the QOS string or by invoking property setter methods. Refer to the HTML layer documentation to find a description of those properties and their default values. The following examples demonstrates parameterization of common protocol objects:

- `"IPMCAST(ttl=2)"` applies a IP multicast time-to-live value of 2. The same can be achieved by calling the method `IPMCAST.setTTL(int ttl)`.
- `"FRAG(size=2048)"` fragments postings into chunks of 2 kilobytes. The fragment size can be set only via the QOS string and cannot be tuned at run-time.
- `"NAK(epochsz=100,retrinterval=4000)"` uses an epoch size of 100 messages and a retransmission interval of 4000 milliseconds.
- `"REACH(timeout=10000)"` to use a time-out of 10 seconds. If REACH has not received any heartbeats from an application during time-out milliseconds, it tags the application as crashed. The same can be achieved by calling the method `REACH.setTimeout(int timeout)`.

## 4-4 Aliases

To make creation of QOS strings easier you can refer to frequently used strings using aliases. Note that aliases can be embedded in QOS strings: for secure and reliable communication one could specify a QOS such as `CRYPT:Reliable`.

Alias	Expands to
Reliable	DISPATCH:PULL:FRAG:FIFO:NAK:REACH:IPMCAST
Unreliable	DISPATCH:IPMCAST
Point-to-Point	DISPATCH:TCP

**Table 2: QOS alias expansion done by the Stack class**

## 4-5 Code Example

```
import iBus.*;
import iBus.layers.*;

// Using QOS strings:
// s1 and s2 implement the same QOS:
Stack s1 = new Stack("Reliable");
Stack s2 = new Stack("DISPATCH:PULL:FRAG:FIFO:NAK:REACH:IPMCAST");

// manually compose a stack:
Stack s3 = new Stack(); // empty QOS, no protocol objects.
s3.attach(new IPMCAST());

// s4 has the same QOS as s3:
Stack s4 = new Stack("IPMCAST");
```



```
// peek at protocol objects:  
REACH r = (REACH)s2.getProtocolObject("REACH");  
FIFO f = (FIFO)s1.getProtocolObject("FIFO");
```

## 5 Postings

➤ See *iBus.Posting*

All data to be pushed or pulled via iBus needs to be encapsulated in **posting** objects. Class `Posting` provides the abstraction of a dynamic array of serializable Java objects. A posting also provides addressing information useful to its receivers.

A posting has a *length* property which is to be set by the `setLength` method. Initially the length is zero. A length of *N* allows you to fill up to *N* objects into a posting. Objects are referred through an integer index. Accessing a posting beyond the value of its length property leads to an exception.

Any `java.io.Serializable` object can be packed into a posting and sent through a channel. This includes your own objects that implement interface `Serializable`, AWT components, Java Beans, many of the JDK classes, iBus postings, protocol objects, and so forth.

On the sending side, an object is serialized by traversing its references to other objects in the object graph recursively to create a complete serialized representation of the graph. On the receiving side the object graph is de-serialized and reconstructed. This default behavior is tailored by using the `transient` Java keyword or by overwriting the JDK methods `writeObject` and `readObject`. For details refer to the JavaSoft documentation on JDK serialization.

Class `Posting` uses an increment factor to speed up applications that call `setLength` repeatedly to increase the length by a small amount. With an increment factor of 2 (the default), calling `setLength` with the current length plus one doubles the size of the internal buffer of the posting.

### 5-1 Code Example

```
Posting p = new Posting();
p.setLength(2);

p.setObject(0, "hello");
p.setObject(1, new java.util.Date());

// this throws an exception since the value of the length
// property is exceeded:
p.setObject(2, "world");

// read a posting:
String hello = (String)p.getObject(0);
java.util.Date d = (java.util.Date)p.getObject(1);

// this throws an exception since the value of the length
// property is exceeded:
String world = (String)p.getObject(2);

// get the length of the posting:
```



```
int length = p.getLength();  
// length is 2.
```

## 6 Pushing and Pulling Postings

⇒ See *iBus.Stack*

The `Stack` class not only serves for interconnecting protocol objects but also as the API by which programmers transmit and receive postings. The class declaration of `iBus.Stack` is as follows:

```
public class Stack {
    public Stack() { ... };
    public Stack(String qos) { ... };

    public void push (iBusURL url, Posting p) { ... };
    public Posting[] pull(iBusURL url, Posting request) { ... };

    public void subscribe(iBusURL url, Receiver rcv) { ... };
    public void unsubscribe(iBusURL url, Receiver rcv) { ... };
    public void registerTalker(iBusURL url) { ... };
    public void unregisterTalker(iBusURL url) { ... };
    ...
};
```

The default constructor ties an empty quality of service string which means no protocol objects are associated with the stack. This kind of stack is needed for adding protocol objects manually as was exemplified in Section 4-5.

A QOS string is passed to create protocol objects using the Java class loader. The remaining methods are used for sending and receiving data, they are explained below.

### 6-1 Pushing Data

To transmit a posting via iBus, the operation

```
void push(iBusURL url, Posting p)
```

is used. `url` denotes the iBus channel through which the posting object is transmitted by the protocol embodied by the stack. Push communication is in one direction from the talker to the listeners, and non-blocking.

### 6-2 Pulling Data

In order to explicitly *request* data from an iBus application, a client application invokes the operation

```
Posting [] pull(iBusURL url, Posting request)
```



Pull communication is two-way and blocking, analogous to RMI. The Java thread that issues a `pull` operation is blocked until the `request` posting is dispatched by the listener objects subscribed to the `URL` and until the sender's protocol stack has received a reply posting from one of the listeners. If no reply is received within a configurable timeout, then zero replies are returned. If all listeners have crashed without returning a reply then an exception is thrown.

iBus always returns the first arriving reply posting, subsequent replies to the pull operation are silently discarded (Figure 2). Note that pull communication is supported only by the qualities of service `Reliable` and `TCP`...

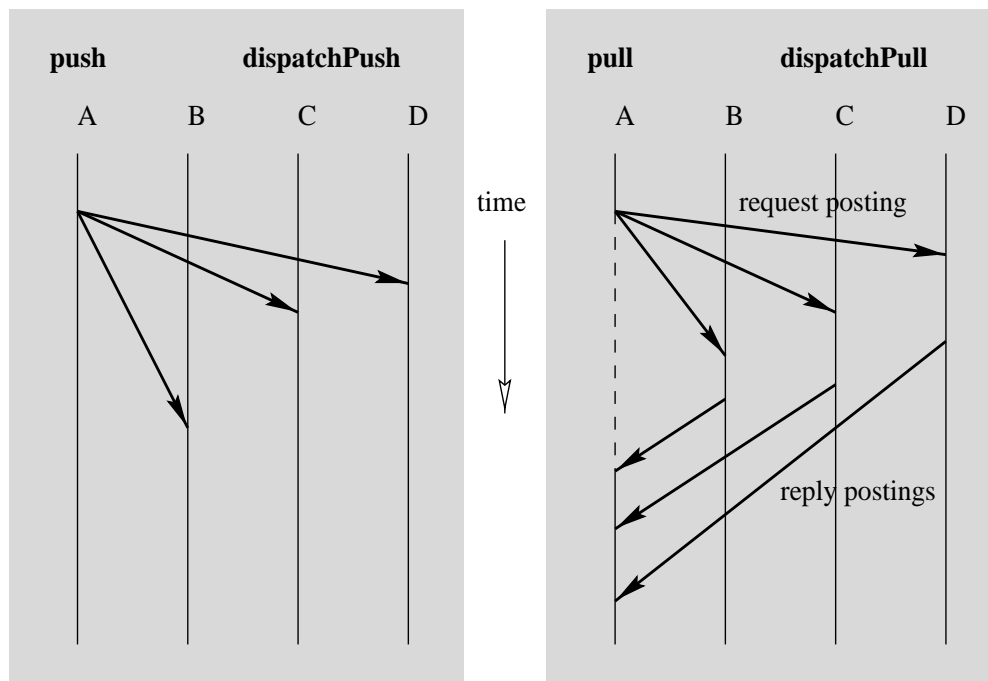


Figure 2: Push and pull communication. A - D denote iBus applications, arrows denote network messages. A push is simply multicast on the channel. Pull blocks the sender (dotted line) until a reply is received.

### 6-3 Opening Transmission Channels

Before you push or pull any postings you need to open the channel by calling the `registerTalker` method. This is a once-only activity for each channel and needed mainly for the management of internal iBus resources. There is also a `unregisterTalker` method for closing a channel. Typically you call `unregisterTalker` when you will not be transmitting on the channel any more.



## 6-4 Subscribing Receiver Objects

➤ See *iBus.Receiver*

For registering an interest in the postings that are pushed and pulled to and from a channel, listener applications issue the

```
void subscribe(iBusURL url, Receiver rcv)
```

operation. The listener application is in charge of providing a Java object (*rcv*) that implements interface `iBus.Receiver`:

```
public interface Receiver {
    void dispatchPush(iBusURL channel, Posting p);
    Posting dispatchPull(iBusURL channel, Posting request);
    void error(iBusURL channel, String details);
};
```

When a push request is received on the channel denoted by `channel`, `iBus` passes the posting along to the `Receiver` object by invoking its `dispatchPush` operation. On a pull request the `dispatchPull` operation is invoked. The posting that is returned by `dispatchPull` is transmitted back to the client that has issued the pull request. If `null` is returned by `dispatchPull` then no reply is returned to the client and it is assumed that another listener will honor the pull request.

An application can subscribe multiple receiver objects to the same channel, and also subscribe the same receiver object to multiple channels.

## 6-5 IP Multicast Communication

IP multicast allows you to transmit datagrams from one sender to many receivers efficiently. `iBus` provides protocol stacks for reliable and unreliable multicast and also provides for transparent fragmentation and reassembly of events that exceed a UDP datagram size.

### 6-5.1 The Basics

The class D IP address range (from 224.0.0.0 to 239.255.255.255) has been reserved by the Internet Assigned Numbers Authority (IANA) for IP multicast<sup>4</sup>. The main differences of a class D address to a point-to-point IP address are that

- an IP multicast address represents a *group* of machines and not just a single machine. Machines can join and leave groups dynamically
- the same class D address can be used on multiple machines
- multicast is used to transport datagrams sent to a IP multicast address
- receivers and senders can be relocated from one machine to another easily
- only UDP is supported and not TCP

---

4. See RFC 1700.



- spontaneous networking is enabled. Applications just need to bind to an IP multicast address to exchange events with all other applications also bound to the address. No name server needs to be consulted.

IP multicast addresses map 1:1 into Ethernet multicast addresses. Thus an information bus which relies on IP multicast provides high scalability and reliability as the dependence on message distribution and naming processes is eliminated.

IP multicast is a unreliable communication protocol meaning that datagrams can get lost or duplicated. There is no such thing as "TCP over IP multicast". An information bus thus needs to provide own reliability protocols.

IP multicast communication is supported by all major operating systems (Windows 95, Windows 98, NT, Solaris, AIX, FreeBSD, Linux etc.). However, as of today IP multicast is available within intranets but not over the whole Internet.

The iBus approach consists in using IP multicast within applications that run in a company's intranet and to forward iBus postings from one intranet to another through a secure TCP connection. Such posting routing software will be provided by SoftWired Inc. in the near future.

#### 6-5.2 Propagation of Datagrams

The propagation of an IP multicast datagram is controlled by the value of its *time to live* (TTL) field. The value can be set through the `tTL` property of the `IPMCAST` protocol object. The value of the TTL field is interpreted as follows.

- 0: postings are delivered only to the applications on the same *host* as the sender
- 1: postings are delivered to the applications on the same *subnet* as the sender. This is the default in iBus.
- 15: postings are delivered to the applications in the same *site* as the sender
- 63: postings are delivered to the applications in the same *region* as the sender
- 127: postings are delivered *worldwide*
- 191: postings are delivered *worldwide*, with limited bandwidth
- 255: no restrictions.

It is up to the routers to impose those restrictions.

## 6-6 TCP Communication

iBus also provides a TCP protocol stack. This stack is intended for iBus applications that communicate via the Internet, for intranet applications requiring only point-to-point channels, and for Applets that get downloaded from the Internet. The TCP stack provides exactly the same push/subscribe API as the multicast stack.

### 6-6.1 Server Applications

A *server application* creates a TCP stack and issues the `subscribe` operation to open a server socket and to start listening for connections. A TCP stack can support multiple TCP channels. The following example code is assumed to run on machine `server.softwired.ch`:

```
// create a TCP stack:
final Stack stack = new Stack("DISPATCH:TCP");

// create a TCP URL with the localhost address and port 7777.
// (With 0 as the port number the next free port is assigned
// to the URL)
final iBusURL url = iBusURLFactory.create("MyServer",
    "1.0", "/requests", "server.softwired.ch", 7777);

MyReceiver rcv = new MyReceiver();
try {
    stack.subscribe(url, rcv);
} catch (Exception e) {
    log_.panic("cannot subscribe to " + url + ": " + e);
}

// wait for connections:
stack.waitTillExit();
```

### 6-6.2 Client Applications

A *client application* connects to a remote TCP stack by issuing the `registerTalker` operation:

```
// create a TCP stack:
final Stack stack = new Stack("DISPATCH:TCP");

// create a TCP URL. The host name denotes the server machine:
final iBusURL url = iBusURLFactory.create("MyServer",
    "1.0", "/requests", "server.softwired.ch", 7777);

try {
    stack.registerTalker(url);
} catch (iBusException e) {
    log_.panic("registerTalker failed: " + e);
}

// now you can transmit postings to the server.
```

### 6-6.3 Client Connect vs. Server Connect

If your firewall set-up does not allow the client to establish a TCP connection to the server you can configure the TCP stack such that `registerTalker` creates the server socket and `subscribe` connects to it. The server application then needs to be modified as follows:

```
// create a TCP stack:
final Stack stack = new Stack("DISPATCH:TCP(listenerconnect=1)");
```



```
// create a TCP URL. The host name denotes the server machine:
final iBusURL url = iBusURLFactory.create("MyServer",
    "1.0", "/requests", "client.softwired.ch", 7777);

MyReceiver rcv = new MyReceiver();
try {
    stack.subscribe(url, rcv);
} catch (Exception e) {
    log_.panic("cannot subscribe to " + url + ": " + e);
}

// wait for connections:
stack.waitTillExit();
```

The client application needs to be modified as follows. Note that now the client needs to be started before the server, and on the machine `server.softwired.ch`:

```
// create a TCP stack:
final Stack stack = new Stack("DISPATCH:TCP(talkerconnect=0)");

// create a TCP URL with the localhost address and port 7777.
// (With 0 as the port number the next free port is assigned
// to the URL)
final iBusURL url = iBusURLFactory.create("MyServer",
    "1.0", "/requests", "client.softwired.ch", 7777);

try {
    stack.registerTalker(url);
} catch (iBusException e) {
    log_.panic("registerTalker failed: " + e);
}

// now you can transmit postings to the server.
```

#### 6-6.4 Multicast via TCP

The TCP stack by default allows multiple talkers to connect with the same listener. The stack can be configured to allow multiple listeners to receive postings from one talker by setting `listenerconnect=1` and `talkerconnect=0` as in the previous example.

### 6-7 Code Example

The next example demonstrates the interaction with a fictive iBus application that implements a time service. To that purpose a `TimeClient` object is provided which employs `pull` communication to retrieve the wall-clock time from a remote time server:

```
import iBus.*;
```

```
public class TimeClient {
    public static void main(String[] argv)
        throws Exception
    {
        // create an iBus protocol stack, a destination URL,
        // and a posting:
        Stack stack = new Stack("Reliable");
        iBusURL url = iBusURLFactory.create("Timeserver",
            "1.0", "/daytime");
        Posting request = new Posting();
        Posting replies[];
        request.setLength(1);
        request.setObject(0, "Get Time");

        // open the channel:
        stack.registerTalker(url);

        // pull a reply posting from a remote time server:
        replies = stack.pull(url, request);

        if (replies.length == 0) {
            // timeout. No reply from server.
            System.err.println("Timeout: no time server");
            // (we could wait and retry the pull operation.)
            return;
        }

        // display the remote time:
        System.out.println("The remote time is "
            + (String)replies[0].getObject(0));

        // close the channel:
        stack.unregisterTalker(url);
    }
}
```



## 7 Membership and Failure Notification

⇒ See *iBus.View*, *iBus.ChannelMember*, *iBus.Membership*,  
*iBus.Stack.registerMonitor()*

In many real-world distributed systems, applications join and leave communication channels in response to external events. An application joins a channel by calling `Stack.subscribe` or `Stack.registerTalker`. An application leaves a channel by calling `Stack.unsubscribe`, `Stack.unregisterTalker`, or by crashing.

iBus provides an API that allows you to register a *membership monitor object* on which iBus invokes a method when applications join or leave a channel you are interested in.

### 7-1 Definition of Terms

- **View:** A view is the “opinion” an application has at a certain time on what members are subscribed to a given channel. Views are represented by the class `iBus.View`.
- **Member:** An application that is subscribed or registered to a channel is called a member. More exactly, not the applications themselves but the protocol stacks in them are the members of the channel. A member is named by an iBus URL.
- **View Change:** When a member joins or leaves a channel, a new `View` object is delivered to the members on the channel. iBus delivers `View` objects by invoking the `viewChange` method on an instance of interface `Membership`.
- **Member Role:** A stack can become a member of a channel in one of three ways: by registering as a listener (`Stack.subscribe`), as a talker (`Stack.registerTalker`), or both. A member can thus be in the role of a listener, of a talker, or both. A `View` object contains the URLs of the members along with their roles.
- **Membership Protocol:** A piece of logic implemented in a protocol object, in REACH for example. The membership protocol makes sure that view changes are delivered to the members of a channel. It can also ensure that all members agree on the views they receive, that push and pull operations are synchronized with view changes, and so forth. Such view synchronization will be available in a future release of iBus.
- **Rank:** A membership protocol may assign a rank number to each member. A rank is an integer value ranging from 0 to  $n - 1$ ,  $n$  being the number of members on the channel. Rank number 0 typically denotes to the oldest member. Rank numbers are contiguous and unique, they are reassigned automatically when a member joins or leaves the channel. This feature is not supported by iBus yet.

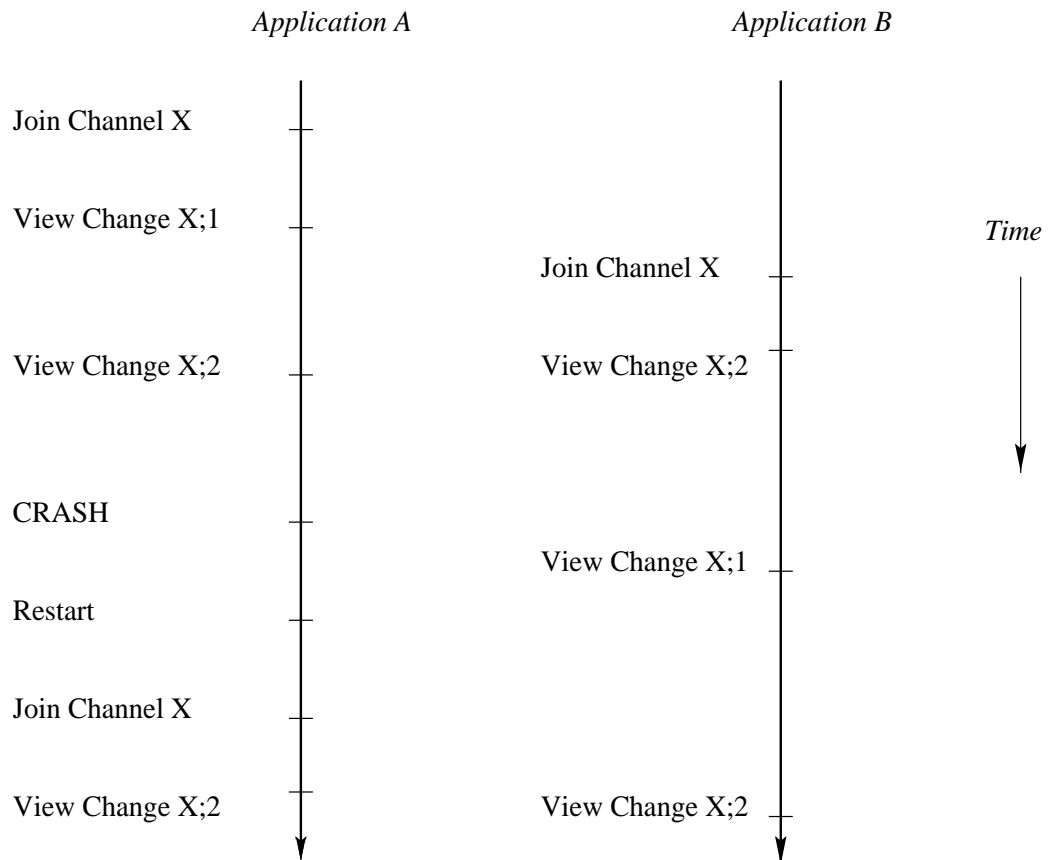


Figure 3: View changes delivered to two iBus applications A and B. First A starts and joins channel X. A view change is delivered to A for X, indicating that there is one channel member. Then B starts and joins channel X. A view change is now delivered to both applications indicating that there are two channel members. A crashes and a view change is delivered to B, indicating that B is the only channel member now. (Now B could initiate a failover protocol). Finally A is restarted and a view change containing two members is delivered to both applications.

## 7-2 Membership Classes

⇒ See *iBus.View*, *iBus.ChannelMember*, *iBus.Membership*, *iBus.Stack.registerMonitor()*

The iBus membership API mainly consists of class `iBus.View`, class `iBus.ChannelMember`, and interface `iBus.Membership`.

A `View` object provides the rank of the member to which the `View` object is passed, through the `getMyRank` accessor. The number of members in the view is obtained by calling `getNumMembers`. `getMember` returns the `ChannelMember` object for a given member URL. The `getMembers` accessor returns an array



of `ChannelMember` objects with one entry per member. `getChannel` returns the channel the view belongs to. Finally, the functions `containsMember`, `containsListener`, and `containsTalker` are used to check whether a specific member exists in a view.

A `iBus.ChannelMember` object mainly provides information on the role of a member through the `isListener` and `isTalker` accessors. `getURL` returns the URL of the member. This URL can be used to compare members, you should not push or subscribe to that URL.

## 7-3 Registering a Membership Monitor

⇒ See *iBus.Membership*, *Stack.registerMonitor*

In order to receive view change notifications you need to provide a class that implements interface `iBus.Membership`. You then pass an instance of the class along to the `Stack.registerMonitor` call. When a view change occurs, the iBus membership protocol object detects it and delivers a view change notification by invoking the `viewChange` method on the `Membership` object you registered with the stack.

Note that to receive view change notifications a protocol stack needs to provide a membership protocol object such as REACH.

## 7-4 Time-Outs

As we saw in Section 4-3 the REACH layer relies on a heartbeat and time-out protocol to detect failures. A heartbeat is transmitted every `hbeat` milliseconds. If the heartbeat of a remote REACH object has not been received for `timeout` milliseconds then the sender is believed crashed and excluded from the view. `hbeat` and `timeout` are tailorable properties of the REACH protocol object.

Those properties are important for a well functioning failure detection, thus some tuning might be required for your application. Typically `hbeat` is set to approximately 1/8 of `timeout`. If the time-out property is low (e.g., < 10.000 milliseconds) then the probability of suspecting an overloaded application as crashed is high. If the time-out value is high (e.g., > 45.000 milliseconds) then the probability of false suspicion is low, but it also takes longer to detect a failure. We recommend using a `timeout` value of 12.000 and a `hbeat` of 1500 for clusters of NT and Unix servers which are moderately loaded. We recommend increasing `timeout` to 45.000 and `hbeat` to 5000 in situations where the network and the machines can become highly loaded, or when Windows 95 PCs are part of the system.

The properties are set as follows:

```
Stack s = new Stack("Reliable");

// obtain the REACH protocol object from the stack:
```



```

REACH r = (REACH)s.getProtocolObject("REACH");
// set timeout and hbeat properties:
r.setTimeout(45000);
r.setHbeat(5000);

```

## 7-5 Code Example

```

import iBus.*;
import iBus.layers.*;

// a class for receiving view changes:
class MyMonitor implements iBus.Membership {
    // upcall method invoked by iBus to deliver a view change:
    public void viewChange(View newView)
    {
        System.err.println("Got a view change for channel: "
            + newView.getChannel() + ". Channel members:");

        // print the URL and the role of each member:
        for (int i = 0; i < newView.getMembers().length; i++) {
            ChannelMember m = newView.getMembers()[i];
            System.err.println(m.getURL()
                + (m.isListener() ? " listener " : "")
                + (m.isTalker() ? " talker" : ""));
        }
    }
}

// register a monitor, then generate view changes by calling
// registerTalker and unregisterTalker repeatedly. Run several
// instances of this application on various machines:
//
public class MembershipTest {
    public static void main(String argv[])
        throws Exception
    {
        // a test channel:
        iBusURL channel
            = iBusURLFactory.create("Test", "1.0", "/test");

        // create a stack containing a membership object (REACH).
        // (The "Reliable" quality of service string would work as
        // well):
        Stack s = new Stack("REACH:IPMCAST");

        // set timeout and heartbeat to low values to make
        // failure detection more aggressive:
        REACH r = (REACH)s.getProtocolObject("REACH");
        r.setTimeout(10000);
        r.setHbeat(1000);

        // create an instance of our membership monitor and register

```



```
// it for the test channel:
MyMonitor mon = new MyMonitor();
s.registerMonitor(channel, mon);

// produce some view changes. iBus will invoke
// mon.viewChange():
for(int i = 0; i < 10; i++) {
    s.registerTalker(channel);
    Thread.currentThread().sleep(5000);
    s.unregisterTalker(channel);
    Thread.currentThread().sleep(15000);
}

// unregister the monitor and terminate the application:
s.unregisterMonitor(channel, mon);
}
}
```

## 8 Logging Events

⇒ See *iBus.Log*

For tracing the execution of an application, iBus provides a simple yet powerful logging facility. The logging facility allows you to assign a severity level to an event, to filter out events that belong to a certain level, to turn on logging just in a specific part of your application, and to direct logging output to a file or to a bus channel.

### 8-1 Log Levels

- `Log.junkLevel`: the most verbose level, used in early development stages of a component. `junkLevel` events are printed only if the log level is set to `Log.junkLevel`.
- `Log.infoLevel`: informative events, statistics etc. `infoLevel` events are printed if the log level is set to either `Log.infoLevel` or `Log.junkLevel`.
- `Log.warnLevel`: for printing warning messages such as a file system filling up. `warnLevel` events are always printed and cannot be turned off. This is level is the default.
- `Log.panicLevel`: for logging unrecoverable errors. `panicLevel` events are always printed and cannot be turned off. The application is aborted by the Log facility.
- `Log.debugLevel`: for tracing problems. This is an alternative to writing directly to `java.lang.System.err`. `debugLevel` events are always printed and cannot be turned off.

Events are logged by passing a string to the methods `Log.junk`, `Log.info`, `Log.warn`, `Log.panic`, and `Log.debug`, respectively.

`Log.setLogLevel` is used for setting the log level. Setting the level to `Log.infoLevel` filters out all events that have a lower level than `infoLevel`, namely `junkLevel` events. Setting it to `Log.warnLevel` filters out `infoLevel` and `junkLevel` events.

By `Log.setDefaultStream` you can specify a JDK `java.io.PrintStream` object to write logging output to. By default `java.lang.System.err` is used.

You typically create one `Log` object per component you want to log. To the `Log` constructor you provide a symbolic name for the component. Now you are able to turn logging on just for that component, by passing its name in conjunction with the `iBusLog` system properties described in Section 8-2. A code example is provided at the end of the chapter.



## 8-2 Logging Events in Java Applications

You can turn logging on by setting the system property `iBusLogLevel` to one of the strings `junk`, `info` or `warn`. This applies to all `Log` objects in the application. By setting the `iBusLog` JDK system property to a colon-delimited list of names you activate logging selectively for certain parts of your application.

The following example shows how to activate `infoLevel` logging just for the REACH and the FRAG protocol object of the `iBus.util.talker` application:

```
% java -DiBusLogLevel=info -DiBusLog="REACH,FRAG" iBus.util.talker
```

The talker application now produces output much like the following:

```
REACH: INFO : startHeartbeat: starting hb thread
REACH: INFO : startHeartbeat: created GroupInfo for
ibus://193.72.83.52:42004/sys/reply
REACH: INFO : run: sending hbeat to
ibus://193.72.83.52:42004/ibus/REACH/hb/sys/reply
REACH: INFO : run: sending hbeat to
ibus://193.72.83.52:42004/ibus/REACH/hb/sys/reply
FRAG: INFO : dnPush: sending a message of 222 bytes
REACH: INFO : run: sending hbeat to
ibus://193.72.83.52:42004/ibus/REACH/hb/sys/reply
```

## 8-3 Redirecting Log Output

By setting the `iBusLogToFile` system property you can direct log output to a file. If the value of `iBusLogToFile` is not defined then a file `ibuslog.txt` is created in the local run directory. A file name can be assigned to the `iBusLogToFile` property. Example:

```
% java -DiBusLogToFile iBus.util.talker
```

```
% java -DiBusLogToFile="mylogfile.txt" iBus.util.talker
```

By setting the `iBusLogToChannel` property the log output is directed to an iBus channel. This allows a system operator to monitor distributed applications by tapping a monitoring GUI tool into the log channel. By default the iBus channel defined by `iBus.Log.LOGCHANNEL` is used. Example:

```
% java -DiBusLogToChannel iBus.util.talker
```

```
% java -DiBusLogToChannel="ibus://226.1.1.2/mylogchannel" \
ibus.util.talker
```

## 8-4 Code Example

The following code fragment shows how the logging facility is used such way that logging can be turned on only for `MyClass` objects.

```
class MyClass {
    private final static Log log_ = new Log("MyClass");

    public void doSomething(String str) {
        log_.info("doSomething: working on " + str);
        ...
        try { ...}
        catch (Exception e) {
            log_.warn("doSomething: error:" + e);
        }
        ...
        log_.info("doSomething: done working on " + str);
    }
}
```

To activate `infoLevel` logging in `MyClass` the application is started by the command

```
% java -DiBusLog=MyClass -DiBusLogLevel=info MyApplication
```



## 9 Threads

⇒ See *iBus.layers.DISPATCH*

iBus provides a variety of threading models for invoking the upcalls defined in interface `iBus.Receiver`. In the simplest case (single-threaded dispatch) the programmer needs to be aware of two threads: the application's `main` thread and the iBus event dispatching thread which invokes `dispatchPush` and `dispatchPull`.

As you may have expected the threading policy is implemented as a protocol object. The DISPATCH protocol object provides the policies

- single-threaded dispatch
- thread-per-request
- thread-pool
- and thread-per-channel.

### 9-1 Single-Threaded Dispatch

Single-threaded dispatch is the default. This means there is exactly one iBus thread *per protocol Stack* which receives incoming messages and other events, and invokes the `dispatchPush` and `dispatchPull` methods.

Only when a dispatch upcall returns the next event can be dispatched. Note that each Stack object contains a dispatcher thread meaning that the upcalls of the receiver objects registered with different stacks run in parallel.

*This policy is suitable when the dispatch operations only perform short computations and do not block for a long time while reading from a database, for example. This model is also to be chosen when the application is subscribed to only one or a few channels and FIFO ordering of requests is important.*

### 9-2 Thread-Per-Request

The performance of servers that carry out long running queries or computations can be improved considerably by using the thread-per-request policy. A new thread is created for each posting to dispatch. When the dispatch operation returns the thread is destroyed. Note that FIFO ordering of incoming messages is no longer guaranteed because postings that arrive on the same channel are dispatched concurrently.

*This policy is for situations where the dispatch upcalls are very long running (many seconds, minutes or more), the event arrival rate is low, and FIFO ordering of events is irrelevant.*

### 9-3 Thread-Pool

A drawback of thread-per-request is that too many threads are created when events arrive at a high rate. Thread-pool works like thread-per-request except that a maximum is imposed on the number of dispatcher threads that are running concurrently. If an event arrives while all threads are in use, the event is buffered until one of the threads becomes free.

*This policy is for situations where the dispatch upcalls are long running, the event arrival rate is high, and FIFO ordering of requests is irrelevant.*

### 9-4 Thread-Per-Channel

This policy is provided to achieve some parallelism while maintaining FIFO ordering of incoming postings. It is conceived for applications that subscribe to several channels. A thread is created for each active subscription. Each thread dispatches the events that arrive on the channel assigned to the thread<sup>5</sup>.

*This policy is for situations where the dispatch upcalls are long running, the application is subscribed to several channels, and FIFO ordering of requests is important. The event arrival rate can be high.*

### 9-5 Comparison

	FIFO ensured	Parallelism	Overhead per event
Single-threaded	Yes	None	Low
Thread-per-request	No	Highest	High
Thread-pool	No	High (customizable)	Low
Thread-per-channel	Yes	Medium	Low

**Table 3: Comparison of the thread policies implemented by DISPATCH**

### 9-6 Code Example

```
// Create a single-threaded TCP Stack (default):
Stack single = new Stack("DISPATCH:TCP");

// Create a thread-per-request TCP Stack:
Stack perReq = new Stack("DISPATCH(threadPerRequest=1):TCP");

// Create a thread-pool TCP Stack with 3 threads:
Stack pool = new Stack("DISPATCH(threadPool=1,poolSize=3):TCP");
```

---

5. With only one channel this model corresponds to single-threaded dispatch.



```
// Create a thread-per-channel TCP Stack with 3 threads:  
Stack perChan = new Stack("DISPATCH(threadPerChannel=1):TCP");  
  
// Create a single-threaded IP multicast Stack (default):  
Stack single2 = new Stack("Reliable");  
  
// Create a thread-per-request IP multicastStack:  
Stack perReq2 = new Stack("DISPATCH(threadPerRe-  
quest=1):PULL:FRAG:FIFO:NAK:REACH:IPMCAST");
```



## Appendix A iBus System Properties

iBus applications are configured by setting the following Java system properties. In general this is accomplished by passing the `-D` option to the Java interpreter. The system properties are accessed by calling `java.lang.System.getProperty()`. You should not modify iBus system properties at run-time.

- `iBusAppName=name` to set the name of the application. The name is used by the logging facility, for example.
- `iBusLogLevel={junk, info, warn}` to set the iBus log level.
- `iBusLog=logmodule1, logmodule2, ...` to activate logging just in certain parts of an application.
- `iBusLogToFile=filename` to direct logging output to a file. If no file name is provided then `ibuslog.txt` is used.
- `iBusLogToChannel=ibusurl` to direct logging output to an iBus channel. If no URL is provided then `iBus.Log.LOGCHANNEL` is used.
- `iBusPort=number` to set the default iBus port number.
- `iBusHostAddress=ipaddress` to set the IP address of the local host. By default `java.net.InetAddress.getLocalHost()` is used. This option is useful when running on machines with multiple IP interfaces.
- `iBusHostName=name` to set the name of the local host. By default `java.net.InetAddress.getLocalHost().getHostName()` is used.
- `iBusSystem=area` to set the name of the system area the application is running in. For example, *production*, *test*, *development*, *joes-test* etc. By default the name `dev` is used.
- `iBusVersion` writes the iBus version string to `System.err`.



## Appendix B A Note about Configuration Management

Companies that provide on-line access to their information system are faced with the problem of running a production information system while developing new versions of the services which constitute the system. Obviously it is necessary to run the production system in isolation of the iBus events that are generated while working on new and improved services.

To this purpose iBus provides the notion of isolated **system areas** allowing one to run a **production** version of a system along with a **test** version of the system and one or more **development** versions of the system. iBus makes sure that the areas are isolated and that the postings of one area do not propagate into other areas.

Ideally the production system runs on a cluster of dedicated machines. Paying customers typically connect to the production system. The most stable versions of your iBus applications are installed on that system. The production system is managed by a crew of operators but is not directly accessible by developers.

The test system can be seen as a "staging area" in which newly developed (and tested) services are installed before they are incorporated into the production system. Ideally the test system is running on a separate set of machines, it is accessible by the development team. Once the test system has been stable for a certain amount of time, the software in the test system is moved to the production system.

When a new service is to be implemented then a development area is used to avoid interfering with the test and, worse, with the production system. Once a certain level of stability is achieved the service is integrated into the test system.

To allow your software to evolve in this manner you

- should always use the `iBusURLFactory` to create iBus URLs. (See Section 3-2)
- set the `iBusSystem` property to the name of the system area your application has to run in.

By default applications run in the `development` system area. To place an iBus application in another area, `production` for example, the command line argument

```
-DiBusSystem=production
```

is passed to the `java` or `jre` command. This sets the JDK system property `iBusSystem` to the value `production`.