PreCache NetInjector Application Development

Contents

PreCache NetInjector Application Development

Welcome to PreCache NetInjector

Tutorials

Basic Case, Publication — Java

Basic Case, Subscription— Java

Common Cases, Publication — Java

Common Cases, Subscription — Java

Basic Case, Publication — C++

Basic Case, Subscription — C++

Common Cases, Publication — C++

Common Cases, Subscription — C++

Basic Case, Publication — C

Basic Case, Subscription — C

Common Cases, Publication — C

Common Cases, Subscription — C

Welcome to PreCache NetInjector

PreCache NetInjector is the industry's first Internet-scale publish/subscribe communication system. Reversing the traditional 'pull' model of content distribution in favor of a subscription-based system, *NetInjector* reduces network load and bandwidth requirements and facilitates meaningful client interactions.

Goals and objectives: *the goals and objectives of this guide*

Conventions: *typefaces for variables, fields, methods, and the like*

NetInjector Overview: *a brief consideration of the basic concepts*

Getting Started: *installing and running the code samples*

Goals and objectives

— the goals and objectives of this guide

PreCache NetInjector Application Development provides basic information about developing applications for the Event Agent.

The API has been packaged in C Types and Functions, C++ Classes, and Java Classes. While each has its advantages and disadvantages for the application developer, each API exposes the same basic functionality.

The goal of this guide is to enable you to write simple applications using typical *NetInjector* programming logic. For more information on the full range of methods and constructors, please see the *PreCache NetInjector API Reference*.

The book presents in a logical, step-by-step format the process of programming the event agent. At least at the beginning, the developer should peruse the guide in order, so as to develop a sense of the overall process of writing *NetInjector*-enabled applications.

Conventions

— typefaces for variables, fields, methods, and the like

The following typeface conventions have been observed in these pages:

- `code`
- *fields*
- links
- *methods*
- `values`
- *variables*

For better readability in both PDF and HTML, these conventions may sometimes print as either boldface or italic.

NetInjector Overview

— a brief consideration of the basic concepts

Push and pull

The traditional publisher posts content to a server and waits for a member of the target audience to find the content and request it.

This 'pull' model is resource intensive and depends on a wide range of complex transactions for success. Not least of these is the requirement that the interested parties find one another in the first place.

Publishing content in a distributed 'push' environment reverses this model: the publisher posts content to a pool of receptive members. There is no need to create redundant content, to manage server farms holding enormous volumes of content, or to facilitate the hapless surfer's search efforts: members of the group receive data automatically, according to the terms of their relationship with the publisher.

> *In creating a stock quote service, the traditional internet news site must store all data related to every stock and post it to everyone who requests the information.*
>
> *The NetInjector news provider can post stock quotes targeted to the specific interests of existing subscribers, rapidly responding to their known preferences.*

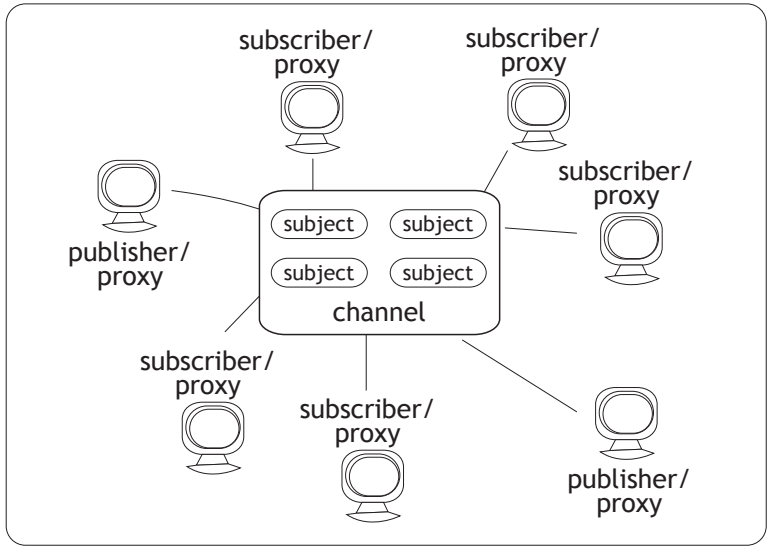
Channels

A key construct of the *NetInjector* solution is the channel. Each channel is a namespace of subjects and attributes. The channel exists as a communications port, facilitating the transfer of data from publisher to subscriber. The channel also assures security and provides stable quality of service.

Each area of interest or content type is a *subject*. Subjects provide a category for content within a channel. Publishers create content for a subject; subscribers receive the subject content.

Channels are further defined by a set of *attributes*, which enhance content distribution. These attributes provide additional flexibility for sorting and routing information and, therefore, packets.

> *The stock service for a news service would be organized as a subject within a NetInjector news channel. The channel's subjects could include, in addition to stocks, other areas of interest such as sports and politics.*
>
> *Within the channel, predefined attributes would allow the publisher and subscriber to filter information by ticker symbol, price, dates, or other data contained within the media provider's information feed.*



Agents

Each client accesses the channel via an agent or *proxy*; the agent is implemented via a proxy. The proxy connection to the router provides an interface for the client to publish and receive notifications for the channel and its subjects.

Proxies receive notifications for the duration of each session. However, the event agent supports methods to pause, suspend, and restart open proxies, avoiding the need to close connections.

Notifications

When a publisher posts new content for a subject within a channel, a notification is prepared for the subscriber base. This notification conforms to the parameters established by the initial configuration of the channel and subject, as well as to the programming logic of the event agent or application.

As the notification passes through the system, the NetInjector router determines the ultimate destination of the notification. At each network node, the notification advances toward the greatest number of subscribers.

Thus, content which targets several subscribers only passes through the network once for a given network node or router. By such *content-based routing*, *NetInjector* routes packets based on content rather than by explicit request, greatly reducing loads.

> *Each notification to a subscriber would contain information about the stock which met the subscriber's particular criteria, consistent with the application logic.*
>
> *Thus, if the application supports filtering by share volume, then a subscriber could ask to be notified when a stock's daily volume reached a certain level of trading activity. So, too, the agent could be programmed to deliver notifications based on price or any other criteria.*

Filtering

Subscribers request content based on their interests. However, the entire data stream for a subject could be vast, exceeding both the requirements of the publisher to serve a target audience and the subscriber's need for content.

By writing filters to better tailor content to a subscriber's interests, the application developer can take best advantage of *NetInjector*'s ability to reduce server load and accelerate the distribution of meaningful content.

These filters may sort data by the predefined criteria established when the channel was created, or they may be created as a concatenation of filters. The application developer may also create filters for specific instances.

> *In delivering notifications, the stock service could allow a subscriber to receive notifications for a given stock once its price exceeded a certain level and trading activity crossed a predetermined barrier.*
>
> *Furthermore, the service could allow the subscriber to create unique filters based on any data within the stream, facilitating notifications by such constructs as moving averages or price/earnings ratios.*

Infrastructure

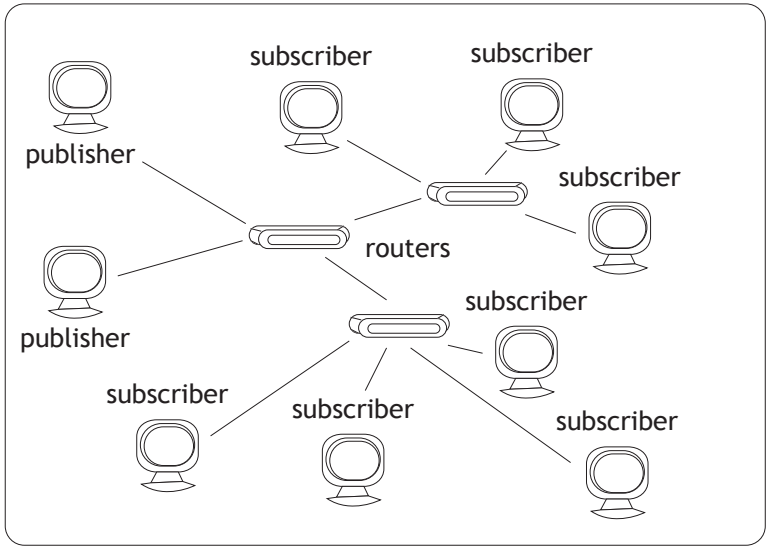
NetInjector supports a two-stage router environment. One stage routes publisher content; the second delivers subscriber content. Together, this simple topology facilitates efficient data flow between the publisher and the subscriber. Because *NetInjector* maintains a logical distinction between stages, no data flows within a stage.

Data flow

In a traditional network, each request from a subscriber results in a unique transaction with the server.

In a *NetInjector* network, the publisher's notification only propagates to those subscribers interested in the content. If all of the subscribers in a neighborhood request the same data, *NetInjector* generates only one notification.

Each data flow is finely tuned to the subjects and attributes of the channel, ensuring that content is distributed to interested subscribers.



The publisher creates content for the channel, creating a data stream for all subscribers. The routers distribute the content based on subscription information. Each notification propagates towards the neighborhood of interested subscribers.

Getting Started

— installing and running the code samples

The sample package of applications presents various ways in which you can publish and receive notifications. The code samples included have been written in Java, C++, and C.

Beyond the samples specifically described in the tutorial, the package includes applications for creating a simple publishing tool, various flavors of chat, and a stock quote service.

Installation and configuration

You will need the following information from your administrator before installing and configuring your sample package:

- the IP addresses of the Primary and Backup NetInjector Information Server
- the *NetInjector* installation's local zip code
- a user id

Windows platforms

1. Dearchive the package into a root directory.
2. Run *setup.bat*.
3. At the prompts, enter the IP addresses, your local installation's zip code, and your user id.
4. Set your *NETINJ_HOME* environment variable to the directory where you dearchived your *NetInjector* sample package.
5. Create a directory for the workspace $*NETINJ_HOME*/samples.
6. Add `$NETINJ_HOME/lib/dynamic` to your path variable.

Linux platforms

1. Dearchive the package into a root directory.
2. Run *setup.bat*.
3. At the prompts, enter the IP addresses, your local installation's zip code, and your user id.
4. Set `NETINJ_HOME` environment variable to the directory where NetInjector is installed.
5. Add `$(NETINJ_HOME)/lib/dynamic` to your `LD_LIBRARY_PATH`.
6. To compile samples, set `SAMPLE_HOME` env variable to `$(NETINJ_HOME)/samples`.
7. To run samples, copy the file *channel.conf* from `$(NETINJ_HOME)/etc/channel.conf` to the sample's directory.

Working with the applications

Each application included in the package is uncompiled.

1. Compile the applications in your workspace.

2. Add the appropriate *channel.conf* file to each application's directory.

   Note: The *channel.conf* file must conform to the parameters of the channel. See, also, the *PreCache NetInjector API Reference* guide, for more information on configuring channels.

3. Run the application.

Tutorials

— an overview of the tutorials

The tutorials included in this guide will help you to build simple applications.

For each application, we write the application logic for both the publisher and subscriber, providing code for Java, C++, and C programmers. Each tutorial builds on the previous exercise.

Tutorial structure: *the basic learning objectives of the tutorials*

Notification Overview: *a high-level view of the notification process*

Channel Overview: *channel configuration, subjects, and attributes*

Tutorial structure

— the basic learning objectives of the tutorials

The tutorials have been organized so as to develop a clear understanding of the basic concepts. Each one introduces a new idea, or develops an existing one.

Note: The tutorials present the basics of writing applications for NetInjector. For more information on each programming language's API, see the *PreCache NetInjector API Reference*.

Basic Case

The first set of tutorials describe the basic process of creating, delivering and receiving notifications.

You will learn the basics of both the publish and subscribe processes from initializing a proxy connection to a channel to publication of a simple notification to the subscriber's retrieval of the notification's content.

Common Cases

In the second tutorial you will learn how to create applications that take advantage of NetInjector's ability to distribute content based on predefined and discretionary attributes.

In learning more about attributes, you will learn how to define and retrieve both predefined and discretionary attributes, and to filter notifications based on attributes and their values.

Notification Overview

— a high-level view of the notification process

Content within a channel is contained in notifications. The distribution of these notifications is at the heart of a publish/subscribe model of content distribution.

Essentially, the notification process entails two procedures:

- A publisher assigns a value to a channel's attribute.
- A subscriber retrieves the value of a channel's attribute.

Publishers assign values to a channel's attributes by invoking the *notify* method or function of a proxy. The proxy establishes and maintains connection to a channel, and validates the publisher's notification against the data type for the attributes.

Subscribers retrieve content in much the same way that publishers create it: they initialize a proxy for the channel and invoke an attribute retrieval process to receive notifications for the subjects within the channel for which they are authorized.

Publishers and subscribers have no direct relationship to one another. Rather, each establishes a proxy connection to a channel and its subjects.

For each subscriber, subjects are captured in a *subscription* or set of subscriptions. The subscription exists for a given subscriber for a particular subject within a channel.

The successful distribution of content ultimately depends on a callback function, in which the application returns data based on the subscription.

Channel Overview

— channel configuration, subjects, and attributes

Channels are a namespace for subjects; content within a channel is organized by subject. Each channel has characteristic attributes.

Proxy connections to *NetInjector* access channels. The event agent, after opening a proxy connection to a channel, publishes and subscribes to the data stream within the channel.

The configuration of the channel must conform to the conventions of the application design for notifications. So, too, the application must conform to the data structure for the channel.

Subjects

Each category of information within a channel is a subject. A *Hobby* channel could have many subjects:

- reading
- arts and crafts
- gardening
- woodworking
- music
- collecting

A subject may have subcategories:

- woodworking
  - woodworking.projects
    - woodworking.projects.interior
    - woodworking.projects.exterior
  - woodworking.wood
    - woodworking.wood.hardwood
    - woodworking.wood.softwood

The subject of our first tutorial was *Sample*. The subcategory was *Intro1*. Thus, the notification will include content for the subject *Sample.Intro1*. The subject of the second tutorial is *Sample.Chat*.

Attributes

Each channel has common attributes called *predefined attributes*, which are created when a channel is committed to the network. Predefined attributes extend across all subjects for a channel.

Attributes are essential to filtering content, whether you are returning simple text messages or developing rich content:

- Attribute filters enable you to create detailed content without building a finely-tuned subject hierarchy.
- Attribute filters facilitate rapid publication of content to narrowly-defined audiences.

Predefined attributes

Predefined attributes may be accessed by the application developer by name or position; predefined attributes are also defined by data type.

Here are a few predefined attributes for the *Hobby* channel:

- experience level
- craft
- material
- message

The predefined attribute referenced in the first tutorial is *Message*. The attributes of Channel 40, which we reference in the second tutorial, include *Message*, *Age*, *Gender*, and *UserName*.

Discretionary attributes

Application designs which require special filtering for a particular subject within a channel must include their own *discretionary attributes*.

Discretionary attributes allow you to deliver content tailored to the needs of the subscriber. If you would like to create attributes unique to a subject or subcategory, you would create discretionary attributes rather than create a new subject for the channel.

Discretionary attributes facilitate the development of unique content and relationships among publishers and subscribers:

- Discretionary attributes enable you to authorize subscriber-defined content.
- Carefully drawn discretionary attributes can enhance performance as you more narrowly target subscribers.

In the second tutorial, we will create one discretionary attribute, *Interest*.

Channel configuration

Each application operates in an environment defined by a channel description. The NetInjector administrator defines the channels' subjects and attributes according to the application's needs, which themselves reflect specific business objectives.

Channel description — *Basic Case*

The channel description for the first tutorial's application defines a channel with two subjects, *Sample* and *Sample.Intro1*. The channel description also defines an attribute for the channel, *Message*, of type PC_*CHARARRAY*.

Channel description — *Common Cases*

The second tutorial's channel description defines a channel with two subjects, *Sample* and *Sample.Chat*. The channel description also defines several attributes for the channel — *Message*, *Age*, *Gender*, and *UserName* — which are of three different data types.

Basic Case, Publication — Java

— creating and distributing content

The publishing process depends on a few simple concepts. In this section, we will describe the basic concepts of Proxy Initialization and Notification, and provide a code sample for the publication of a simple notification.

This chapter also introduces the construct of a simple filter.

Proxy Initialization: *opening a proxy connection to a channel*

Notification: *the basic message unit*

Content publication: *posting notifications to the channel*

Basic Case, Publication code — Java: *the full code sample for publishing*

Proxy Initialization

— opening a proxy connection to a channel

In order to publish content for a channel, the publisher needs to open a proxy session.

The proxy maintains an open connection to the channel, providing access to the channel's subjects and validation against the channel's notification type.

To initialize a proxy session for a client accessing our channel, we implement the following constructor:

```
proxy = new Proxy(channel id, client id)
```

For proxy p, our *channel id* is 7 and our *client id* is 3001:

```
p = new Proxy(7,3002);
```

Note: *client id* represents the id of the customer entered by your administrator into the *User Management* interface of the *Management Console*.

Notification

— the basic message unit

Published content takes the form of a notification; each publishing event is captured as a notification.

For this sample, the notification contains the contents of channel 7's predefined attribute *Message*; the data type of the attribute is *string*. The string contains the content of the notification.

Each notification must access the proxy so that the proxy can validate the notification against the channel's data type for the attribute. The publisher's proxy creates the notification for a particular subject.

```
n = new Notification(proxy, "subject");
```

The notification must define its content. In this case, we define our content by setting a value for a predefined attribute:

```
n.setSetPredefinedAttr("attribute name",
   "attribute value");
```

Here's the code to create a notification for channel 7's subscribers:

```
n = new Notification(p,"Sample.Intro1");
n.setPredefinedAttr("Message","Hello World");
```

Content publication

— posting notifications to the channel

Publishing the notification is the final step in creating content for a channel.

The method for publishing content is:

`proxy.publish(`*`notification`*`);`

To publish our proxy's notification, we invoke the publish method:

`p.publish(n);`

All authorized subscribers who have initialized proxies ready to receive content for Channel 7's subject, *Sample*, will receive the message "Hello World."

Basic Case, Publication code — Java

— the full code sample for publishing

The previous code samples presented the basics of publishing. A publication application initializes channels, proxies, subjects, and attributes for content distribution.

The basic sequence of the publication process in the completed application is:

1. Create a proxy connection for the channel. (Proxy Initialization)
2. Create a notification for the subject; set the value of the predefined attribute. (Notification)
3. Publish the notification for the subject over the channel. (Content publication)

```
import com.precache.event.*;
import java.io.*;

public class pub1
{
    static
   {
      System.loadLibrary("event-j");
   }

   public static void main(String argv[])
   {
//Create a proxy for the channel and client.
      Proxy p;
      Notification n;
      try
      {
```

```
         p = new Proxy(7,3002);

//Create a notification for the subject.
         n = new Notification(p,”Sample.Intro1”);

//Set the value of the predefined attribute.
         n.setPredefinedAttr(
            “Message”,”Hello World”);

//Publish the notification for the subject.
         p.publish(n);
      }

      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

In the next chapter, Basic Case, Subscription— Java, we will construct a simple application for retrieving the published message.

Basic Case, Subscription— Java

— retrieving basic notifications

In the chapter Basic Case, Publication — Java, we created a simple mechanism for publishing content. In this section, we will complete the content distribution cycle by enabling subscribers to retrieve the content.

Subscriber: *the publisher's audience*

Subscription: *the relationship between publisher and subscriber*

Subscription closure: *associating the subscription with a closure*

Notification: *accessing the notification's content*

Basic Case, Subscription code— Java: *the complete subscription code*

Subscriber

— the publisher's audience

In order to retrieve a notification, the application must define a subscriber class. The *notify* method is called whenever a notification matches the subscription.

The class *MySubscriber* overwrites the *Notify* method from the base class *Subscriber*:

```
class MySubscriber implements Subscriber
{
   public void notify(Notification n,
      Object closure)
};
```

***MySubscriber*:** application-specific class that inherits *Subscriber* class

n: a pointer to the newly received notification

closure: the closure associated with the subscription for which the notification should be delivered

Subscription

— the relationship between publisher and subscriber

The proxy must create a subscription requesting the content published for the subject in our channel.

```
proxy.subscribe("subject", new subscriber object,
   closure);
```

For our subscriber, we would like to access the Sample subject created in Basic Case, Publication — Java. After initializing a *Proxy* object p, we invoke our instance of MySubscriber to create a subscription for the subject *Sample.Intro1*.

```
p.subscribe("Sample.Intro1", new MySubscriber(),
   null);
```

Subscription closure

— associating the subscription with a closure

The *subscribe* method sets value of the closure originally defined as a string (in this case), `My First Subscription`.

```
p.subscribe("Sample.Intro1", new MySubscriber(),
   "My First Subscription");
```

We defined this final stage of content retrieval when we created our subscriber class definition:

```
public void notify(Notification n, Object closure)
```

Upon notification delivery, the proxy passes the closure as the second argument of the *notify* method. The value of *closure* is defined by the Subscriber class.

```
System.out.println(
   "Incoming notification for (" + closure + ")");
```

The output of the subscription mapped to the new subscriber is:

```
Incoming notification for (My First Subscription):
```

Notification

— accessing the notification's content

When we published the notification, we set a value for a predefined attribute called *Message*. The subscriber must access this attribute of the channel.

The class definition creating the subscriber already included a reference to the notification:

```
notify(Notification n,Object closure)
```

Each matching notification is passed to the application as the first argument of the *notify* method. *n* contains the contents of the notification's attributes.

The value of our predefined attribute *Message* was set to `Hello World` by the publisher (see Notification, Basic Case, Publication — Java).

getPredefinedAttr retrieves the attribute value stored in the notification. This retrieval occurs within the *notify* method.

```
message = n.getPredefinedAttrString(
   "Message");
System.out.println(
   "Incoming notification for (
   " + closure + ")");
```

The complete subscriber output should read:

```
Incoming notification for (My First Subscription):
   Hello World.
```

Basic Case, Subscription code— Java

— the complete subscription code

The previous code samples presented the basics of subscription. A subscription application retrieves content in much the same way as it was published, albeit after first defining a subscription class.

Here is the basic sequence of the subscription process:

1. Create a proxy connection for the channel. (Proxy Initialization, Basic Case, Publication — Java)
2. Define a subscriber class. (Subscriber)
3. Create the subscription. (Subscription)
4. Associate a closure with the subscription. (Subscription closure)
5. Retrieve the content of the notification. (Notification)

```
import com.precache.event.*;
import java.io.*;

//Define a subscriber class.
class MySubscriber implements Subscriber
{
   public MySubscriber()
   {
   }
//Define the callback method.
   public void notify(Notification n,
      Object closure)
   {
      String message;
      try
      {
//Retrieve the value of the predefined attribute.
         message = n.getPredefinedAttrString(
   "Message");
         System.out.println(
         "Incoming notification for (
         " + closure + "):" + message);
      }
      catch(Exception e)
      {
         System.out.println(e);
      }
   }
};

public class sub1
{
   static
   {
      System.loadLibrary("event-j");
   }

   public static void main(String argv[])
   {
//Create a proxy for the channel and client.
   Proxy p;
   try
   {
      p = new Proxy(7,3002);

//Submit the subscription.
      p.subscribe("Sample.Intro1",
         new MySubscriber(),
         "My First Subscription");

      System.out.println(
         "Subscription was submitted...
```

```
            waiting 30 seconds for incoming
            notifications”);
            Thread.currentThread().sleep(30000);
         }

         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
}
```

In the next chapter, Common Cases, Publication — Java, we will learn more about attributes.

Common Cases, Publication — Java

— assigning attribute values

In our first tutorial, we published a simple notification. The notification's content was the value of a predefined attribute of a channel.

In this tutorial, we will develop the concept of attributes to create rich content.

Notifications with predefined attributes: *assigning values to predefined attributes*

Notifications with discretionary attributes: *assigning values to discretionary attributes*

Common Cases, publication code — Java: *the full code sample for the Common Cases tutorial*

Notifications with predefined attributes

— assigning values to predefined attributes

When publishing content, you construct your notification by assigning value to predefined attributes. In the first tutorial, we set a value for one predefined attribute of Channel 7, *Message*.

In the *Sample.Chat* subject of Channel 40, we set values for four predefined attributes: *Message*, *Username*, *Age*, and *Gender*. These four predefined attributes and their values are of three different data types:

Message, Username: PC_STRING

Age: PC_LONG

Gender: PC_CHAR

The data types for these attributes were set when the *NetInjector* administrator configured the channel.

To set the values for these attributes, we call the *Notification* class's *setPredefinedAttr* method.

```
n1.setPredefinedAttr("UserName","Jane");
n1.setPredefinedAttr("Age",10);
n1.setPredefinedAttr("Gender",'F');
```

Note: For each attribute, we set the attribute by name. Attribute values may also be set by position.

Notifications with discretionary attributes

— assigning values to discretionary attributes

Sometimes you may want to allow subscribers to develop content or relationships to groups beyond the limits of the predefined attributes. In such cases, create discretionary attributes.

To create a discretionary attribute, call the *Notification* class's *addDiscretionaryAttr* method when you construct your notification:

```
n2.addDiscretionaryAttr("Interest","Country Music");
```

The data type of our discretionary attribute, *Interest*, is *PC_STRING*.

Note: *addDiscretionaryAttr* always references the attribute's *name*. However, the data type of the attribute may be any standard data type.

Common Cases, publication code — Java

— the full code sample for the Common Cases tutorial

The sample for this tutorial presented more examples of setting attributes. While the process of publication is the same as in Basic Case, Publication — Java, we have added statements for

- Notifications with predefined attributes and for
- Notifications with discretionary attributes.

```
import com.precache.event.*;
import java.io.*;

public class pub2
{
   static
   {
      System.loadLibrary(“event-j”);
   }

   public static void main(String argv[])
   {
      Proxy p;
      Notification n1,n2;

      try
      {
//Create a proxy for the channel and client.
         p = new Proxy(40,3002);

//Create a notification.
         n1 = new Notification(p,”Sample.Chat”);

//Set the values for the predefined attributes.
         n1.setPredefinedAttr(“Message”,
            “Make this world a better place!”);
         n1.setPredefinedAttr(“UserName”,”Jane”);
         n1.setPredefinedAttr(“Age”,10);
         n1.setPredefinedAttr(“Gender”,'F');

//Publish the notification.
         p.publish(n1);

//Create a second notification.
         n2 = new Notification(p,”Sample.Chat”);

//Set the values for the predefined attributes.
         n2.setPredefinedAttr(“Message”,
            “Dolly Parton, I love you!”);
         n2.setPredefinedAttr(“UserName”,
            “Laurel”);
         n2.setPredefinedAttr(“Age”,12);
         n2.setPredefinedAttr(“Gender”,'f');

//Set the value of a discretionary attribute.
         n2.addDiscretionaryAttr(“Interest”,
            “Country Music”);

//Publish the second notification.
         p.publish(n2);
      }

      catch (Exception e)
      {
         e.printStackTrace();
      }
   }
}
```

In the final tutorial chapter, Common Cases, Subscription — Java, we will build the participating subscriber’s application.

Common Cases, Subscription — Java

— retrieving attribute values

In the chapter Common Cases, Publication — Java, we created a mechanism for publishing notifications with both predefined and discretionary attribute values. In this section, we enable subscribers to retrieve this content.

In this tutorial, we will focus on retrieving filtered content.

Retrieving predefined attributes: *retrieving predefined attributes of various types*

Attribute filters: *defining subscriptions*

Discretionary attributes: *filtering on discretionary attributes*

Retrieving discretionary attributes: *retrieving discretionary attribute values*

Common Cases, subscription code — Java: *the complete subscription code*

Retrieving predefined attributes

— retrieving predefined attributes of various types

The subscriber class definition for the first tutorial included a single predefined attribute, *Message*:

```
message = n.getPredefinedAttrString("Message");
```

The new subscriber class definition includes three more predefined attributes: *UserName*, *Age*, and *Gender*:

```
username = n.getPredefinedAttrString("UserName");
age = n.getPredefinedAttrLong("Age");
gender = n.getPredefinedAttrChar("Gender");
```

In the first part of this tutorial, Common Cases, Publication — Java, we set the values for these predefined attributes.

Attribute filters

— defining subscriptions

All subscriptions filter content for at least one attribute. That attribute may be as simple as *Message*, or may be based on complex subscriber input.

In the first tutorial, we retrieved all attribute values associated with a channel for a particular subject. We can, however, also filter a notification's content.

To filter a notification's content for a particular predefined attribute, we assign it a value. In this example, we have assigned a value to the *UserName* attribute.

```
p.subscribe("Sample.Chat","UserName==\"Jane\",
   new Subscriber1(),"Listen to Jane");
```

The subscriber might want to receive notifications published by other young women in the channel. Here, our subscriber could choose to filter notifications for *Gender* and *Age*.

```
p.subscribe("Sample.Chat",
   "((Gender == 'F' or Gender == 'f') and Age < 14)",
   new Subscriber1(),"Girl Scouts");
```

Note that the values in this subscription must conform to the data types set for the channel's attributes.

Discretionary attributes

— filtering on discretionary attributes

If our subscriber would like to define a specific area of interest, the application would need to be enable her to create a discretionary attribute. We created this discretionary attribute when we defined Notifications with discretionary attributes in the previous exercise.

To create the subscription for the discretionary attribute, we add the filter to the subscription:

```
p.subscribe(“Sample.Chat”,
   “Interest == \”Country Music\”Music\””,
   new Subscriber1(),“Country music fans”);
```

Because the program created Common Cases, Publication — Java supports a discretionary attribute for *Interest*, the subscriber can refine her subscription to reflect her interest in *Country Music*.

Retrieving discretionary attributes

— retrieving discretionary attribute values

In accessing the data for a discretionary attribute, we relied on the publisher’s definition of the attribute. By filtering on that specific value, our subscriber could retrieve the content of notifications matching her criteria.

We can also enhance the subscription so that the proxy retrieves the values assigned to the discretionary attribute.

```
n.getDiscretionaryAttr(“Interest”);
```

getDiscretionaryAttr adds the value of the discretionary attribute *Interest* to the subscription. The value of this attribute has already been set by the publisher as *Country Music*. (Common Cases, Publication — Java)

So as to call *getDiscretionaryAttr*, we add the statement above to a new instance of the Subscriber class, *Subscriber2*.

When the proxy creates the new subscription, including the filter for the discretionary attribute *Interest*, the *subscribe* method invokes *Subscriber2*.

```
p.subscribe(“Sample.Chat”,
   “Interest == \”Country Music\”Music\””,
   new Subscriber2(),“Country music fans”);
```

Our new subscriber definition enables the subscriber to retrieve content more tightly coupled to her interests.

Common Cases, subscription code — Java

— the complete subscription code

The previous code samples presented a simple publish/subscribe application. In this tutorial, we have learned how to enhance subscriptions by

- retrieving predefined attributes;
- filtering notifications using attribute filters (for either predefined or discretionary attributes); and
- retrieving discretionary attributes.

```
import com.precache.event.*;
import java.io.*;

//Define a subscriber.
class Subscriber1 implements Subscriber
{
   public Subscriber1()
   {
   }
   public void notify(Notification n,
      Object closure)
   {
      String message;
      try
      {
         String username;
         char   gender;
         long   age;

//Retrieve the value of the predefined attributes.
         message = n.getPredefinedAttrString(
            "Message");
         username = n.getPredefinedAttrString(
            "UserName");
         age = n.getPredefinedAttrLong("Age");
         gender = n.getPredefinedAttrChar(
            "Gender");

//Build the notification.
         System.out.println("Incoming notification
            for (" + closure + "):" + message);
         System.out.println("From:");
         System.out.println("Name = " + username);
         System.out.println("Age  = " + age);

         if ((gender == 'M')||(gender=='m'))
            System.out.println("Male");
         else
         if ((gender == 'F')||(gender=='f'))
            System.out.println("Female");
            System.out.println();

      }

      catch(Exception e)
      {
         System.out.println(e);
      }
   }
};

//Define a subscriber which can retrieve a
//discretionary attribute.
   public Subscriber2()
   {
   }
   public void notify(Notification n,
      Object closure)
```

```
   {
      String message;
      try
      {
         String username;
         char   gender;
         long   age;
         String interest;

//Retrieve the value of the predefined attributes.
         message = n.getPredefinedAttrString(
            "Message");
         username = n.getPredefinedAttrString(
            "UserName");
         age = n.getPredefinedAttrLong("Age");
         gender = n.getPredefinedAttrChar(
            "Gender");

//Retrieve the value of the discretionary attribute.
         n.getDiscretionaryAttr("Interest");

         System.out.println("Incoming notification
            for (" + closure + "):" + message);
         System.out.println("From:");
         System.out.println("Name = " + username);
         System.out.println("Age  = " + age);

         if ((gender == 'M')||(gender=='m'))
            System.out.println("Male");
         else
         if ((gender == 'F')||(gender=='f'))
            System.out.println("Female");
            System.out.println("Interest =
               " + interest);

            System.out.println();
         }

         catch(Exception e)
         {
            System.out.println(e);
         }
      }
};


public class sub2
{
   static
   {
      System.loadLibrary("event-j");
   }

   public static void main(String argv[])
   {
//Create a proxy for the channel and client.
      Proxy p;
      try
      {
         p = new Proxy(40,3002);

//Submit the subscription.
         p.subscribe("Sample.Chat",
            "UserName==\"Jane\"UserName==\"Jane\"",
         new Subscriber1(),
            "Listen to Jane");

         p.subscribe("Sample.Chat",
            "((Gender == 'F' or Gender == 'f')
            and Age < 14)",
            new Subscriber1(),
            "Girl Scouts");
```

```
            p.subscribe("Sample.Chat",
               "Interest == \"Country Music\"Music\"",
               new Subscriber2(),
               "Country music fans");

            System.out.println("Subscription was
               submitted... waiting 30 seconds for
               incoming notifications");
            Thread.currentThread().sleep(30000);
         }
         catch (Exception e)
         {
            e.printStackTrace();
         }
      }
   }
```

Basic Case, Publication — C++

— creating and distributing content

The publishing process depends on a few simple concepts. In this section, we will describe the basic concepts of Proxy Initialization and Notification, and provide a code sample for the publication of a simple notification.

This chapter also introduces the construct of a simple filter.

Proxy Initialization: *opening a proxy connection to a channel*

Notification: *the basic message unit*

Content publication: *posting notifications to the channel*

Basic Case, Publication code — C++: *the full code sample for publishing*

Proxy Initialization

— opening a proxy connection to a channel

In order to publish content for a channel, the publisher needs to open a proxy session.

The proxy maintains an open connection to the channel, providing access to the channel's subjects and validation against the channel's notification type.

To initialize a proxy session for a client accessing our channel, we implement the following constructor:

```
Proxy (channel id, client id)
```

For proxy p, our *channel id* is 7 and our *client id* is 3001:

```
Proxy p(7,3001);
```

Note: *client id* represents the id of the customer entered by your administrator into the *User Management* interface of the *Management Console*.

Notification

— the basic message unit

Published content takes the form of a notification; each publishing event is captured as a notification.

For this sample, the notification contains the contents of channel 7's predefined attribute *Message*; the data type of the attribute is *string*. The string contains the content of the notification.

Each notification must access the proxy so that the proxy can validate the notification against the channel's data type for the attribute. The publisher's proxy creates the notification for a particular subject.

```
Notification n(proxy, "subject");
```

The notification must define its content. In this case, we define our content by setting a value for a predefined attribute:

```
n.setSetPredefinedAttr(
   "attribute name","attribute value");
```

Here's the code to create a notification for channel 7's subscribers:

```
Notification n(p,"Sample.Intro1");
n.SetPredefinedAttr("Message","Hello World");
```

Content publication

— posting notifications to the channel

Publishing the notification is the final step in creating content for a channel.

The method for publishing content is:

`proxy.Publish(`*`notification`*`);`

To publish our proxy's notification, we invoke the publish method:

```
p.Publish(n);
```

All authorized subscribers who have initialized proxies ready to receive content for Channel 7's subject, *Sample*, will receive the message "Hello World."

Basic Case, Publication code — C++

— the full code sample for publishing

The previous code samples presented the basics of publishing. A publication application initializes channels, proxies, subjects, and attributes for content distribution.

The basic sequence of the publication process in the completed application is:

1. Create a proxy connection for the channel. (Proxy Initialization)
2. Create a notification for the subject; set the value of the predefined attribute. (Notification)
3. Publish the notification for the subject over the channel. (Content publication)

```
#include "PC_evn_Filter.hpp"
#include "PC_evn_Proxy.hpp"
#include <iostream>

using namespace precache::event;
using std::cerr;
using std::endl;

int
main(int argc, char* argv[])
{
   try
   {

//Create a proxy for the channel and client.
      Proxy p(7,3001);
```

```
//Create a notification for the subject.
      Notification n(p,”Sample.Intro1”);

//Set the value of the predefined filter.
      n.SetPredefinedAttr(“Message”,”Hello World”);

//Publish the notification for the subject.
      p.Publish(n);

   } catch (PCException &e)
   {
      cerr<<e.Message().String()<<endl;
   }
}
```

In the next chapter, Basic Case, Subscription — C++, we will construct a simple application for retrieving the published message.

Basic Case, Subscription — C++

— retrieving basic notifications

In the chapter *Basic Case, Publication — C++*, we created a simple mechanism for publishing content. In this section, we will complete the content distribution cycle by enabling subscribers to retrieve the content.

Subscriber: *the publisher's audience*

Subscription: *the relationship between publisher and subscriber*

Subscription closure: *associating the subscription with a closure*

Notification: *accessing the notification's content*

Basic Case, Subscription code — C++: *the complete subscription code*

Subscriber

— the publisher's audience

In order to retrieve a notification, the application must define a subscriber class. The *Notify* method is called whenever a notification matches the subscription.

The class *MySubscriber* overwrites the *Notify* method from the base class *Subscriber*:

```
class MySubscriber:public Subscriber
{
public:
   virtual void Notify(
   Notification* pNotification,
   void* pClosure)
   throw (precache::util::PCException)
   {
      cout<<"Incoming notification"<<endl;
   }
};
```

MySubscriber: application-specific class that inherits *Subscriber* class

pNotification: a pointer to the newly received notification

pClosure: the closure associated with the subscription for which the notification should be delivered

Subscription

— the relationship between publisher and subscriber

The proxy must create a subscription requesting the content published for the subject in our channel.

```
proxy.Subscribe("subject", new subscriber object,
   closure);
```

For our subscriber, we would like to access the Sample subject created in Basic Case, Publication — C++. After initializing a *Proxy* object p, we invoke our instance of MySubscriber to create a subscription for the subject *Sample.Intro1*.

```
p.Subscribe("Sample.Intro1", new MySubscriber(),
   NULL);
```

Subscription closure

— associating the subscription with a closure

The *Subscribe* method sets value of the `NULL` closure originally defined as a string (in this case), `My First Subscription`.

```
p.Subscribe("Sample.Intro1", new MySubscriber(),
   strdup("My First Subscription"));
```

We defined this final stage of content retrieval when we created our subscriber class definition:

```
Notify(Notification* pNotification, void* pClosure)
```

Upon notification delivery, the proxy passes the closure as the second argument of the *Notify* method. The value of *pClosure* is defined by the Subscriber class.

```
cout<<"Incoming notification for
   ("<<(char*)pClosure<<")"<<endl;
```

The output of the subscription mapped to the new subscriber is:

```
Incoming notification for My First Subscription:
```

Notification

— accessing the notification's content

When we published the notification, we set a value for a predefined attribute called *Message*. The subscriber must access this attribute of the channel.

The class definition creating the subscriber already included a reference to the notification:

```
Notify(Notification* pNotification, void* pClosure)
```

Each matching notification is passed to the application as the first argument of the *Notify* method. *pNotification* contains the contents of the notification's attributes.

The value of our predefined attribute *Message* was set to `Hello World` by the publisher (see Notification, Basic Case, Publication — C++).

GetPredefinedAttr retrieves the attribute value stored in the notification. This retrieval occurs within the *Notify* method.

```
pNotification->GetPredefinedAttr("Message",message);
cout<<"Incoming notification for
   ("<<(char*)pClosure<<"):"<<message<<endl;
```

The complete subscriber output should read:

```
Incoming notification for My First Subscription:
   Hello World.
```

Basic Case, Subscription code — C++

— the complete subscription code

The previous code samples presented the basics of subscription. A subscription application retrieves content in much the same way as it was published, albeit after first defining a subscription class.

Here is the basic sequence of the subscription process:

1. Create a proxy connection for the channel. (Proxy Initialization, Basic Case, Publication — C++)
2. Define a subscriber class. (Subscriber)
3. Create the subscription. (Subscription)
4. Associate a closure with the subscription. (Subscription closure)
5. Retrieve the content of the notification. (Notification)

```
#include “PC_evn_Filter.hpp”
#include “PC_evn_Proxy.hpp”
#include “PC_evn_Notification.hpp”
#include “PC_evn_Subscriber.hpp”
#include <iostream>

using namespace precache::event;
using std::cerr;
using std::cout;
using std::endl;

//Define a subscriber class.
class MySubscriber:public Subscriber
{
   public:

//Define the callback method.
   virtual void Notify(
      Notification* pNotification,
      void* pClosure)
      throw (precache::util::PCException)
   {
      PC_STRING message;

//Retrieve the value of the predefined attribute.
      pNotification->GetPredefinedAttr(
         “Message”,message);
      cout<<“Incoming notification for (
         “<<(char*)pClosure<<“):
         “<<message<<endl;
   }
};

int
main(int argc, char* argv[])
{
   try
   {

//Create a proxy for the channel and client.
      Proxy p(7,3002);

//Submit the subscription.
      p.Subscribe(
         “Sample.Intro1”,new MySubscriber(),
         strdup(“My First Subscription”));

      cout<<“Subscription was submitted...
      waiting 30 seconds for incoming
      notifications”
      <<endl;

      sleep(30);
```

```
    }

    catch (PCException &e)
    {
        cerr<<e.Message().String()<<endl;
    }
}
```

In the next chapter, Common Cases, Publication — C++, we will learn more about attributes.

Common Cases, Publication — C++

— assigning attribute values

In our first tutorial, we published a simple notification. The notification's content was the value of a predefined attribute of a channel.

In this tutorial, we will develop the concept of attributes to create rich content.

Notifications with predefined attributes: *assigning values to predefined attributes*

Notifications with discretionary attributes: *assigning values to discretionary attributes*

Common Cases, publication code — C++: *the full code sample for the Common Cases tutorial*

Notifications with predefined attributes

— assigning values to predefined attributes

When publishing content, you construct your notification by assigning value to predefined attributes. In the first tutorial, we set a value for one predefined attribute of Channel 7, *Message*.

In the *Sample.Chat* subject of Channel 40, we set values for four predefined attributes: *Message*, *Username*, *Age*, and *Gender*. These four predefined attributes and their values are of three different data types:

Message, Username: PC_STRING

Age: PC_LONG

Gender: PC_CHAR

The data types for these attributes were set when the *NetInjector* administrator configured the channel.

To set the values for these attributes, we call the *Notification* class's *SetPredefinedAttr* method.

```
n1.SetPredefinedAttr("UserName","Jane");
n1.SetPredefinedAttr("Age",(PC_LONG)10);
n1.SetPredefinedAttr("Gender",'F');
```

Note: For each attribute, we set the attribute by name. Attribute values may also be set by position.

Notifications with discretionary attributes

— assigning values to discretionary attributes

Sometimes you may want to allow subscribers to develop content or relationships to groups beyond the limits of the predefined attributes. In such cases, create discretionary attributes.

To create a discretionary attribute, call the *Notification* class's *AddDiscretionaryAttr* method when you construct your notification:

```
n2.AddDiscretionaryAttr("Interest","Country Music");
```

The data type of our discretionary attribute, *Interest*, is *PC_STRING*.

Note: *AddDiscretionaryAttr* always references the attribute's *name*. However, the data type of the attribute may be any standard data type.

Common Cases, publication code — C++

— the full code sample for the Common Cases tutorial

The sample for this tutorial presented more examples of setting attributes. While the process of publication is the same as in Basic Case, Publication — C++, we have added statements for

- Notifications with predefined attributes and for
- Notifications with discretionary attributes.

```
#include “PC_evn_Filter.hpp”
#include “PC_evn_Proxy.hpp”
#include <iostream>

using namespace precache::event;
using std::cerr;
using std::endl;

int
main(int argc, char* argv[])
{
   try
   {
//Create a proxy for the channel and client.
      Proxy p(40,3001);

//Create a notification.
      Notification n1(p,”Sample.Chat”);

//Set the values for predefined attributes.
      n1.SetPredefinedAttr(
         “Message”,”Hello world!”);
      n1.SetPredefinedAttr(“UserName”,”Jane”);
      n1.SetPredefinedAttr(“Age”,(PC_LONG)10);
      n1.SetPredefinedAttr(“Gender”,'F');

//Publish the notification.
      p.Publish(n1);

//Create a second notification.
      Notification n2(p,”Sample.Chat”);

//Set the values for predefined attributes.
      n2.SetPredefinedAttr(“Message”,”Dolly
Parton, I love you!”);
      n2.SetPredefinedAttr(“UserName”,”Laurel”);
      n2.SetPredefinedAttr(“Age”,(PC_LONG)12);
      n2.SetPredefinedAttr(“Gender”,'f');

//Set a discretionary attribute’s value.
      n2.AddDiscretionaryAttr(“Interest”,”Country
Music”);

//Publish the second notification.
      p.Publish(n2);

   }

      catch (PCException &e)
      {
         cerr<<e.Message().String()<<endl;
      }
}
```

In the final tutorial chapter, Common Cases, Subscription — C++, we will build the participating subscriber’s application.

Common Cases, Subscription — C++

— retrieving attribute values

In the chapter *Common Cases, Publication — C++*, we created a mechanism for publishing notifications with both predefined and discretionary attribute values. In this section, we enable subscribers to retrieve this content.

In this tutorial, we will focus on retrieving filtered content.

Retrieving predefined attributes: *retrieving predefined attributes of various types*

Attribute filters: *defining subscriptions*

Discretionary attributes: *filtering on discretionary attributes*

Retrieving discretionary attributes: *retrieving discretionary attribute values*

Common Cases, subscription code — C++: *the complete subscription code*

Retrieving predefined attributes

— retrieving predefined attributes of various types

The subscriber class definition for the first tutorial included a single predefined attribute, *Message*:

```
pNotification->GetPredefinedAttr("Message",message);
```

The new subscriber class definition includes three more predefined attributes: *UserName*, *Age*, and *Gender*:

```
pNotification->GetPredefinedAttr("Name",name);
pNotification->GetPredefinedAttr("Age",age);
pNotification->GetPredefinedAttr("Gender",gender);
```

In the first part of this tutorial, Common Cases, Publication — C++, we set the values for these predefined attributes.

Attribute filters

— defining subscriptions

All subscriptions filter content for at least one attribute. That attribute may be as simple as *Message*, or may be based on complex subscriber input.

In the first tutorial, we retrieved all attribute values associated with a channel for a particular subject. We can, however, also filter a notification's content.

To filter a notification's content for a particular predefined attribute, we assign it a value. In this example, we have assigned a value to the *UserName* attribute.

```
p.Subscribe("Sample.Chat","UserName==\"Jane\",
   new Subscriber1(),strdup("Listen to Jane"));
```

The subscriber might want to receive notifications published by other young women in the channel. Here, our subscriber could choose to filter notifications for *Gender* and *Age*.

```
p.Subscribe("Sample.Chat","(
   (Gender == 'F' or Gender == 'f') and Age <14)",
   new Subscriber1(),strdup("Girl Scouts"));
```

Note that the values in this subscription must conform to the data types set for the channel's attributes.

Discretionary attributes

— filtering on discretionary attributes

If our subscriber would like to define a specific area of interest, the application would need to be enable her to create a discretionary attribute. We created this discretionary attribute when we defined Notifications with discretionary attributes in the previous exercise.

To create the subscription for the discretionary attribute, we add the filter to the subscription:

```
p.Subscribe("Sample.Chat","Interest == \"Country
    Music\"Music\"",new Subscriber1(),strdup("Country
    music fans"));
```

Because the program created Common Cases, Publication — C++ supports a discretionary attribute for *Interest*, the subscriber can refine her subscription to reflect her interest in *Country Music*.

Retrieving discretionary attributes

— retrieving discretionary attribute values

In accessing the data for a discretionary attribute, we relied on the publisher's definition of the attribute. By filtering on that specific value, our subscriber could retrieve the content of notifications matching her criteria.

We can also enhance the subscription so that the proxy retrieves the values assigned to the discretionary attribute.

```
pNotification->GetDiscretionaryAttr("Interest",
    (void*&)interest,typecode,size);
```

GetDiscretionaryAttr adds the value of the discretionary attribute *Interest* to the subscription. The value of this attribute has already been set by the publisher as *Country Music*. (Common Cases, Publication — C++)

So as to call *GetDiscretionaryAttr*, we add the statement above to a new instance of the Subscriber class, *Subscriber2*.

When the proxy creates the new subscription, including the filter for the discretionary attribute *Interest*, the *Subscribe* method invokes *Subscriber2*.

```
p.Subscribe("Sample.Chat","Interest == \"Country
    Music\"Music\"",new Subscriber2(),strdup("Country
    music fans"));
```

Our new subscriber definition enables the subscriber to retrieve content more tightly coupled to her interests.

Common Cases, subscription code — C++

— the complete subscription code

The previous code samples presented a simple publish/subscribe application. In this tutorial, we have learned how to enhance subscriptions by

- retrieving predefined attributes;
- filtering notifications using attribute filters (for either predefined or discretionary attributes); and
- retrieving discretionary attributes.

```
#include “PC_evn_Filter.hpp”
#include “PC_evn_Proxy.hpp”
#include “PC_evn_Notification.hpp”
#include “PC_evn_Subscriber.hpp”
#include <iostream>

using namespace precache::event;
using std::cerr;
using std::cout;
using std::endl;

//Define a subscriber.
class Subscriber1:public Subscriber
{
public:
   virtual void Notify(
   Notification* pNotification,
   void* pClosure)
   throw (precache::util::PCException)
   {
      PC_STRING message;
      PC_STRING username;
      PC_CHAR   gender;
      PC_LONG   age;

//Retrieve the values of predefined attributes.
      pNotification->GetPredefinedAttr(
         “Message”,message);
      pNotification->GetPredefinedAttr(
         “UserName”,username);
      pNotification->GetPredefinedAttr(“Age”,age);
      pNotification->GetPredefinedAttr(“Gender”,gender);

//Build the notification.
      cout<<“Incoming notification for (
         “<<(char*)pClosure<<“):”<<message<<endl;
      cout<<“From:”<<endl;
      cout<<“Name = “<< username <<endl;
      cout<<“Age  = “<< (int)age <<endl;
         if ((gender == 'M')||(gender=='m'))
            cout<<“Male”<<endl;
            else
               if ((gender == 'F')||(gender=='f'))
                  cout<<“Female”<<endl;
                  cout<<endl;
         }
   };

//Define a subscriber which can retrieve a
//discretionary attribute.
class Subscriber2:public Subscriber
{
public:
   virtual void Notify(
      Notification* pNotification,
      void* pClosure)
      throw (precache::util::PCException)
   {
```

```
        PC_STRING message;
        PC_STRING username;
        PC_CHAR   gender;
        PC_LONG   age;
        PC_STRING interest;
        PC_TypeCode typecode;
        PC_UINT size;

//Retrieve the value of the predefined attributes.
   pNotification->GetPredefinedAttr(
      “Message”,message);
   pNotification->GetPredefinedAttr(
      “UserName”,username);
   pNotification->GetPredefinedAttr(“Age”,age);
   pNotification->GetPredefinedAttr(“Gender”,gender);
   pNotification->

//Retrieve the value of the discretionary attribute.
   GetDiscretionaryAttr(
      “Interest”,
      (void*&)interest,
      typecode,size);

   cout<<“Incoming notification for (
      “<<(char*)pClosure<<“):”<<message<<endl;


//Build the notification.
   cout<<“From:”<<endl;
   cout<<“Name = “<<username<<endl;
   cout<<“Age  = “<<(int)age<<endl;
   if ((gender == 'M')||(gender=='m'))
      cout<<“Male”<<endl;
      else
         if ((gender == 'F')||(gender=='f'))
            cout<<“Female”<<endl;
            cout<<“Interest = “<<interest<<endl;
            cout<<endl;
   }
};

int
main(int argc, char* argv[])
{
   try
   {
//Create a proxy for the channel and client.
      Proxy p(40,3002);

//Submit the subscription.
      p.Subscribe(

“Sample.Chat”,”UserName==\”Jane\”Sample.Chat”,”UserN
ame==\”Jane\””, new
         Subscriber1(),
         strdup(“Listen to Jane”));

      p.Subscribe(
         “Sample.Chat”,
         “((Gender == 'F' or Gender == 'f')
            and Age < 14)”,
         new Subscriber1(), strdup(“Girl Scouts”));

      p.Subscribe(
         “Sample.Chat”,
         “Interest == \”Country Music\”Music\””,
         new Subscriber2(),
         strdup(“Country music fans”));

      cout<<“Subscriptions were submitted...
         waiting 30 seconds for incoming
         notifications”
         <<endl;
```

```
            sleep(30);
        }
        catch (PCException &e)
        {
            cerr<<e.Message().String()<<endl;
        }

    }
```

Basic Case, Publication — C

— creating and distributing content

The publishing process depends on a few simple concepts. In this section, we will describe the basic concepts of Proxy Initialization and Notification, and provide a code sample for the publication of a simple notification.

This chapter also introduces the construct of a simple filter.

Proxy Initialization: *opening a proxy connection to a channel*

Notification: *the basic message unit*

Content publication: *posting notifications to the channel*

Basic Case, Publication code — C: *the full code sample for publishing*

Proxy Initialization

— opening a proxy connection to a channel

In order to publish content for a channel, the publisher needs to open a proxy session.

The proxy maintains an open connection to the channel, providing access to the channel's subjects and validation against the channel's notification type.

To initialize a proxy session for a client accessing our channel, we implement the following function:

```
PC_evn_create_proxy(proxy,channel id,client id);
```

For proxy p, our *channel id* is 7 and our *client id* is 3001:

```
PC_evn_create_proxy(&p,7,3001);
```

Note: *client id* represents the id of the customer entered by your administrator into the *User Management* interface of the *Management Console*.

Notification

— the basic message unit

Published content takes the form of a notification; each publishing event is captured as a notification.

For this sample, the notification contains the contents of channel 7's predefined attribute *Message*; the data type of the attribute is *string*. The string contains the content of the notification.

Each notification must access the proxy so that the proxy can validate the notification against the channel's data type for the attribute. The publisher's proxy creates the notification for a particular subject.

```
PC_evn_create_notification(notification,
   proxy,"subject",closure);
```

The notification must define its content. In this case, we define our content by setting a value for a predefined attribute:

```
PC_evn_set_predefined_attr_by_name(notification,
   "Attribute","attribute value",
   data type,closure));
```

Here's the code to create a notification for channel 7's subscribers:

```
PC_evn_create_notification(&n,
   p,"Sample.Intro1",NULL);

PC_evn_set_predefined_attr_by_name(n,
   "Message","Hello World",
   PC_STRING_TYPE,strlen("Hello World"));
```

Content publication

— posting notifications to the channel

Publishing the notification is the final step in creating content for a channel.

The function for publishing content is:

`PC_evn_publish(`*`proxy`*`,`*`notification`*`);`

To publish our proxy's notification, we invoke the publish function:

```
PC_evn_publish(p,n);
```

All authorized subscribers who have initialized proxies ready to receive content for Channel 7's subject, *Sample*, will receive the message "Hello World."

Basic Case, Publication code — C

— the full code sample for publishing

The previous code samples presented the basics of publishing. A publication application initializes channels, proxies, subjects, and attributes for content distribution.

The basic sequence of the publication process in the completed application is:

1. Create a proxy connection for the channel. (Proxy Initialization)
2. Create a notification for the subject; set the value of the predefined attribute. (Notification)
3. Publish the notification for the subject over the channel. (Content publication)

```
#include <stdio.h>
#include "PC_utl_log.h"
#include "PC_evn_filter.h"
#include "PC_evn_proxy.h"
#include "PC_evn_notification.h"

int
main(int argc, char* argv[])
{
   PC_Status nStatus;

//Create a proxy for the channel and client.
   PC_evn_Proxy p;
   PC_evn_Notification n;

   nStatus = PC_evn_create_proxy(&p,7,3001);
```

```
   if (nStatus != PC_SUCCESS)
   {
      printf("Proxy creation has failed\n");
      return 0;
   }

//Create a notification for the subject.
   nStatus = PC_evn_create_notification(
      &n, p,"Sample.Intro1",NULL);

   if (nStatus != PC_SUCCESS)
   {
      printf("Notification creation has failed
         \n");
      return 0;
   }

//Set the value of the predefined attribute.
   PC_evn_set_predefined_attr_by_name(n,
      "Message","Hello World",
      PC_STRING_TYPE,strlen("Hello World"));

//Publish the notification for the subject.
   nStatus = PC_evn_publish(p,n);

   if (nStatus != PC_SUCCESS)
   {
      printf("Failed to publish notification
         \n");
      return 0;
   }

}
```

In the next chapter, Basic Case, Subscription — C, we will construct a simple application for retrieving the published message.

Basic Case, Subscription — C

— retrieving basic notifications

In the chapter Basic Case, Publication — C, we created a simple mechanism for publishing content. In this section, we will complete the content distribution cycle by enabling subscribers to retrieve the content.

Callback Function: *the publisher's audience*

Subscription: *the relationship between publisher and subscriber*

Subscription closure: *associating the subscription with a closure*

Notification: *accessing the notification's content*

Basic Case, Subscription code — C: *the complete subscription code*

Callback Function

— the publisher's audience

In order to retrieve a notification, the application must define a callback function. The callback function is called whenever a notification matches the subscription.

```
void notify(PC_evn_Notification notification,
   void* pClosure)
{
   printf("Incoming notification \n");
};
```

notification: the newly received notification

pClosure: the closure associated with the subscription for which the notification should be delivered

Subscription

— the relationship between publisher and subscriber

The proxy must create a subscription requesting the content published for the subject in our channel.

```
PC_evn_subscribe_with_exp(proxy,
   "subject",PC_NULL,
   callback function,closure);
```

For our subscriber, we would like to access the Sample subject created in Basic Case, Publication — C. After initializing a *Proxy* object p, we invoke our callback function, *notify*, to create a subscription for the subject *Sample.Intro1*.

```
PC_evn_subscribe_with_exp(p,
   "Sample.Intro1",PC_NULL,
   notify,PC_NULL,
   PC_NULL, &hSubscription);
```

Subscription closure

— associating the subscription with a closure

The *Subscribe* function sets value of the `NULL` closure originally defined as a string (in this case), `My First Subscription`.

```
PC_evn_subscribe_with_exp(p,
   “Sample.Intro1”,PC_NULL,
   notify,strdup(“My First Subscription”),
   PC_NULL, &hSubscription);
```

We defined this final stage of content retrieval when we created our callback function:

```
notify(PC_evn_Notification notification, void*
   pClosure)
```

Upon notification delivery, the proxy passes the closure as the second argument of the *notify* function. The value of *pClosure* is defined by the Callback Function.

```
printf(“Incoming notification for (%s):%s\n”,
   (char*)pClosure);
```

The output of the subscription is:

```
Incoming notification for My First Subscription:
```

Notification

— accessing the notification’s content

When we published the notification, we set a value for a predefined attribute called *Message*. The subscriber must access this attribute of the channel.

The callback function already included a reference to the notification:

```
void notify(PC_evn_Notification notification,
   void* pClosure)
```

Each matching notification is passed to the application as the first argument of the *notify* function. *notification* contains the contents of the notification’s attributes.

The value of our predefined attribute *Message* was set to `Hello World` by the publisher (see Notification, Basic Case, Publication — C).

PC_evn_get_predefined_attr_by_name retrieves the attribute value stored in the notification. This retrieval occurs within the *PC_evn_Notification* function.

```
PC_evn_get_predefined_attr_by_name(
   notification,”Message”,&message,
   PC_STRING_TYPE,&size);
```

```
printf(“Incoming notification for (%s):%s\n”,
   (char*)pClosure,message);
```

The complete subscriber output should read:

```
Incoming notification for (My First Subscription):
   Hello World.
```

Basic Case, Subscription code — C

— the complete subscription code

The previous code samples presented the basics of subscription. A subscription application retrieves content in much the same way as it was published, albeit after first defining a subscription class.

Here is the basic sequence of the subscription process:

1. Create a proxy connection for the channel. (Proxy Initialization, Basic Case, Publication — C)
2. Define a subscriber class. (Callback Function)
3. Create the subscription. (Subscription)
4. Associate a closure with the subscription. (Subscription closure)
5. Retrieve the content of the notification. (Notification)

```
#include <stdio.h>
#include "PC_utl_log.h"
#include "PC_evn_filter.h"
#include "PC_evn_proxy.h"
#include "PC_evn_notification.h"

//Define a callback function.
void notify(PC_evn_Notification notification,
   void* pClosure)
{
   PC_STRING message;
   PC_UINT size;

//Retrieve the value of the predefined attribute.
   PC_evn_get_predefined_attr_by_name(
      notification,
      "Message",&message,
      PC_STRING_TYPE,&size);

   printf("Incoming notification for (%s):%s\n",
      (char*)pClosure,message);
};

int
main(int argc, char* argv[])
{
   PC_Status nStatus;

//Create a proxy for the channel and client.
   PC_evn_Proxy p;
   PC_evn_Subscription_Handle hSubscription;

   nStatus = PC_evn_create_proxy(&p,7,3001);

   if (nStatus != PC_SUCCESS)
   {
      printf("Proxy creation has failed\n");
      return 0;
   }

//Submit the subscription.
   nStatus = PC_evn_subscribe_with_exp(
      p,"Sample.Intro1",
      PC_NULL,
      notify,
      strdup("My First Subscription"),
      PC_NULL, &hSubscription);

   if (nStatus != PC_SUCCESS)
   {
      printf("Subscription submission has failed
         \n");
```

```
        return 0;
    }
        printf("Subscription was submitted
        ... waiting 30 seconds for incoming
        notifications\n");

    sleep(30);
}
```

In the next chapter, Common Cases, Publication — C, we will learn more about attributes.

Common Cases, Publication — C

— assigning attribute values

In our first tutorial, we published a simple notification. The notification's content was the value of a predefined attribute of a channel.

In this tutorial, we will develop the concept of attributes to create rich content.

Notifications with predefined attributes: *assigning values to predefined attributes*

Notifications with discretionary attributes: *assigning values to discretionary attributes*

Common Cases, publication code — C: *the full code sample for the Common Cases tutorial*

Notifications with predefined attributes

— assigning values to predefined attributes

When publishing content, you construct your notification by assigning value to predefined attributes. In the first tutorial, we set a value for one predefined attribute of Channel 7, *Message*.

In the *Sample.Chat* subject of Channel 40, we set values for four predefined attributes: *Message*, *Username*, *Age*, and *Gender*. These four predefined attributes and their values are of three different data types:

Message, Username: PC_STRING

Age: PC_LONG

Gender: PC_CHAR

The data types for these attributes were set when the *NetInjector* administrator configured the channel.

To set the values for these attributes, we call the *Notification* class's *PC_evn_set_predefined_attr_by_name* method.

```
PC_evn_set_predefined_attr_by_name(n1,
    “UserName”,”Jane”,
    PC_STRING_TYPE,strlen(“Jane”));

PC_evn_set_predefined_attr_by_name(n1,
    “Age”,&age,PC_LONG_TYPE,0);

PC_evn_set_predefined_attr_by_name(n1,
    “Gender”,&gender,
    PC_CHAR_TYPE,0);
```

Note: For each attribute, we set the attribute by name. Attribute values may also be set by position.

Notifications with discretionary attributes

— assigning values to discretionary attributes

Sometimes you may want to allow subscribers to develop content or relationships to groups beyond the limits of the predefined attributes. In such cases, create discretionary attributes.

To create a discretionary attribute, call the *Notification* class's *PC_evn_add_discretionary_attr* method when you construct your notification:

```
PC_evn_add_discretionary_attr(n2,
    “Interest”,”Country Music”,
    PC_STRING_TYPE,strlen(“Country Music”));
```

The data type of our discretionary attribute, *Interest*, is *PC_STRING*.

Note: *PC_evn_add_discretionary_attr* always references the attribute's *name*. However, the data type of the attribute may be any standard data type.

Common Cases, publication code — C

— the full code sample for the Common Cases tutorial

The sample for this tutorial presented more examples of setting attributes. While the process of the publication is the same as in Basic Case, Publication — C, we have added statements for

- Notifications with predefined attributes and for
- Notifications with discretionary attributes.

```
#include <stdio.h>
#include “PC_utl_log.h”
#include “PC_evn_filter.h”
#include “PC_evn_proxy.h”
#include “PC_evn_notification.h”

main(int argc, char* argv[])
{
   PC_Status Status;
   PC_evn_Proxy p;
   PC_evn_Notification n1,n2;
   PC_LONG age;
   PC_CHAR gender;

//Create a proxy for the channel and client.
   nStatus = PC_evn_create_proxy(&p,40,3001);

   if (nStatus != PC_SUCCESS)
   {
      printf(“Proxy creation has failed\n”);
      return 0;
   }

//Create a notification.
   nStatus = PC_evn_create_notification(
      &n1,
      p
      “Sample.Chat”,NULL);
   if (nStatus != PC_SUCCESS)
   {
      printf(“Notification creation has failed
         \n”);
      return 0;
   }

//Set the values for predefined attributes.
   PC_evn_set_predefined_attr_by_name(
      n1,”Message”,
      “Make this world a better place!”,
      PC_STRING_TYPE,
      strlen(“Make this world a better place!”));

      PC_evn_set_predefined_attr_by_name(
      n1,
      “UserName”,
      “Jane”,
      PC_STRING_TYPE,
      strlen(“Jane”));

   age = 10;
   PC_evn_set_predefined_attr_by_name(
      n1,
      “Age”,
      &age,PC_LONG_TYPE,0);

   gender = 'F';
   PC_evn_set_predefined_attr_by_name(
      n1,
      “Gender”,
      &gender,
      PC_CHAR_TYPE,0);
```

```
//Publish the notification.
   nStatus = PC_evn_publish(p,n1);
   if (nStatus != PC_SUCCESS)
   {
      printf("Failed to publish notification
         \n");
      return 0;
   }

//Create a second notification.
   nStatus = PC_evn_create_notification(
      &n2,
      p,
      "Sample.Chat",NULL);

   if (nStatus != PC_SUCCESS)
   {
      printf("Notification creation has failed
         \n");
      return 0;
   }

//Set the values for predefined attributes.
   PC_evn_set_predefined_attr_by_name(
      n2,"Message",
      "Dolly Parton, I love you!",
       PC_STRING_TYPE,
      strlen("Dolly Parton, I love you!"));

   PC_evn_set_predefined_attr_by_name(
      n2,"UserName",
      "Laurel",
      PC_STRING_TYPE,
      strlen("Laurel"));

   age = 12;

   PC_evn_set_predefined_attr_by_name(
      n2,"Age",&age,
      PC_LONG_TYPE,0);

   gender = 'f';
   PC_evn_set_predefined_attr_by_name(
      n2,"Gender",&gender,
      PC_CHAR_TYPE,0);

//Set a discretionary attribute's value.
   PC_evn_add_discretionary_attr(
      n2,"Interest","Country Music",
      PC_STRING_TYPE,strlen("Country Music"));

//Publish the second notification.
   nStatus = PC_evn_publish(p,n2);
   if (nStatus != PC_SUCCESS)
   {
      printf("Failed to publish notification
         \n");
      return 0;
   }

}
```

In the final tutorial chapter, Common Cases, Subscription — C, we will build the participating subscriber's application.

Common Cases, Subscription — C

— retrieving attribute values

In the chapter Common Cases, Publication — C, we created a mechanism for publishing notifications with both predefined and discretionary attribute values. In this section, we enable subscribers to retrieve this content.

In this tutorial, we will focus on retrieving filtered content.

Retrieving predefined attributes: *retrieving predefined attributes of various types*

Attribute filters: *defining subscriptions*

Discretionary attributes: *filtering on discretionary attributes*

Retrieving discretionary attributes: *retrieving discretionary attribute values*

Common Cases, subscription code — C: *the complete subscription code*

Retrieving predefined attributes

— retrieving predefined attributes of various types

The subscriber class definition for the first tutorial included a single predefined attribute, *Message*:

```
PC_evn_get_predefined_attr_by_name(notification,
    “Message”,&message,PC_STRING_TYPE,&size);
```

The new subscriber class definition includes three more predefined attributes: *UserName*, *Age*, and *Gender*:

```
PC_evn_get_predefined_attr_by_name(notification,
    “UserName”,&username,PC_STRING_TYPE,&size);
PC_evn_get_predefined_attr_by_name(notification,
    “Age”,&age,PC_LONG_TYPE,&size);
PC_evn_get_predefined_attr_by_name(notification,
    “Gender”,&gender,PC_CHAR_TYPE,&size);
```

In the first part of this tutorial, Common Cases, Publication — C, we set the values for these predefined attributes.

Attribute filters

— defining subscriptions

All subscriptions filter content for at least one attribute. That attribute may be as simple as *Message*, or may be based on complex subscriber input.

In the first tutorial, we retrieved all attribute values associated with a channel for a particular subject. We can, however, also filter a notification’s content.

To filter a notification’s content for a particular predefined attribute, we assign it a value. In this example, we have assigned a value to the *UserName* attribute.

```
PC_evn_subscribe_with_exp(
    p,”Sample.Chat”,
    ”UserName==\”Jane\””,
    notify1, strdup(“Listen to Jane”),
    PC_NULL, &hSubscription1);
```

The subscriber might want to receive notifications published by other young women in the channel. Here, our subscriber could choose to filter notifications for *Gender* and *Age*.

```
PC_evn_subscribe_with_exp(
    p,”Sample.Chat”,
    “((Gender == 'F' or Gender == 'f') and Age < 14)”,
    notify1,strdup(“Girl Scouts”),
    PC_NULL, &hSubscription2);
```

Note that the values in this subscription must conform to the data types set for the channel’s attributes.

Discretionary attributes

— filtering on discretionary attributes

If our subscriber would like to define a specific area of interest, the application would need to be enable her to create a discretionary attribute. We created this discretionary attribute when we defined Notifications with discretionary attributes in the previous exercise.

To create the subscription for the discretionary attribute, we add the filter to the subscription:

```
PC_evn_subscribe_with_exp(
   p,"Sample.Chat",
   "Interest == \"Country Music\"Music\"",
   notify1,strdup("Country music fans"),
   PC_NULL, &hSubscription3);
```

Because the program created Common Cases, Publication — C supports a discretionary attribute for *Interest*, the subscriber can refine her subscription to reflect her interest in *Country Music*.

Retrieving discretionary attributes

— retrieving discretionary attribute values

In accessing the data for a discretionary attribute, we relied on the publisher's definition of the attribute. By filtering on that specific value, our subscriber could retrieve the content of notifications matching her criteria.

We can also enhance the subscription so that the proxy retrieves the values assigned to the discretionary attribute.

```
PC_evn_get_discretionary_attr(notification,
   "Interest",&interest,&typecode,&size);
```

PC_evn_get_discretionary_attr adds the value of the discretionary attribute *Interest* to the subscription. The value of this attribute has already been set by the publisher as *Country Music*. (Common Cases, Publication — C)

So as to call *PC_evn_get_discretionary_attr*, we add the statement above to a new callback function, *notify2*.

When the proxy creates the new subscription, including the filter for the discretionary attribute *Interest*, *PC_evn_subscribe_with_exp* invokes *notify2*.

```
PC_evn_subscribe_with_exp(
   p,"Sample.Chat",
   "Interest == \"Country Music\"Music\"",
   notify2,strdup("Country music fans"),
   PC_NULL, &hSubscription3);
```

Our new subscriber definition enables the subscriber to retrieve content more tightly coupled to her interests.

Common Cases, subscription code — C

— the complete subscription code

The previous code samples presented a simple publish/subscribe application. In this tutorial, we have learned how to enhance subscriptions by

- retrieving predefined attributes;
- filtering notifications using attribute filters (for either predefined or discretionary attributes); and
- retrieving discretionary attributes.

```
#include <stdio.h>
#include “PC_utl_log.h”
#include “PC_evn_filter.h”
#include “PC_evn_proxy.h”
#include “PC_evn_notification.h”

//Define a callback function.
void notify1(
   PC_evn_Notification notification,
   void* pClosure)
{
   PC_STRING message;
   PC_STRING username;
   PC_CHAR   *gender;
   PC_LONG   *age;
   PC_UINT size;

//Retrieve the values of predefined attributes.
   PC_evn_get_predefined_attr_by_name(
      notification,
      “Message”,&message,
      PC_STRING_TYPE,&size);

   PC_evn_get_predefined_attr_by_name(
      notification,
      “UserName”,&username,
      PC_STRING_TYPE,&size);

   PC_evn_get_predefined_attr_by_name(
      notification,
      “Age”,&age,
      PC_LONG_TYPE,&size);

   PC_evn_get_predefined_attr_by_name(
      notification,
      “Gender”,&gender,
      PC_CHAR_TYPE,&size);

//Build the notification.
   printf(“Incoming notification for
      (%s):%s\n”,
      (char*)pClosure,
      message);
   printf(“From:\n”);
   printf(“Name = %s\n”,username);
   printf(“Age  = %d\n”,(int)*age);
   if ((*gender == 'M')||(*gender=='m'))
      printf(“Male\n”) ;
   else
      if ((*gender == 'F')||(*gender=='f'))
      printf(“Female\n”);

};
//Define a callback function which can retrieve a
//discretionary attribute.
void notify2(
   PC_evn_Notification notification,
   void* pClosure)
```

```
{
   PC_STRING message;
   PC_STRING username;
   PC_CHAR *gender;
   PC_LONG *age;
   PC_UINT size;
   PC_TypeCode typecode;
   PC_STRING interest;

//Retrieve the value of the predefined attributes.
   PC_evn_get_predefined_attr_by_name(
      notification,
      “Message”,&message,
      PC_STRING_TYPE,&size);

   PC_evn_get_predefined_attr_by_name(
      notification,
      “UserName”,&username,
      PC_STRING_TYPE,&size);

   PC_evn_get_predefined_attr_by_name(
      notification,
      “Age”,&age,
      PC_LONG_TYPE,&size);

   PC_evn_get_predefined_attr_by_name(
      notification,
      “Gender”,&gender,
      PC_CHAR_TYPE,&size);

//Retrieve the value of the discretionary attribute.
   PC_evn_get_discretionary_attr(
      notification,
      “Interest”,&interest,
      &typecode,&size);

//Build the notification.
   printf(“Incoming notification for
      (%s):%s\n”,
      (char*)pClosure,message);
   printf(“From:\n”);
   printf(“Name = %s\n”,username);
   printf(“Age  = %d\n”,(int)*age);
   if ((*gender == 'M')||(*gender=='m'))
      printf(“Male\n”) ;
   else
      if ((*gender == 'F')||(*gender=='f'))
         printf(“Female\n”);

   printf(“Interest = %s\n\n”,interest);
};

int
main(int argc, char* argv[])
{
   PC_Status nStatus;
   PC_evn_Proxy p;
   PC_evn_Subscription_Handle
      hSubscription1,
      hSubscription2,
      hSubscription3;

//Create a proxy for the channel and client.
   nStatus = PC_evn_create_proxy(&p,40,3001);

   if (nStatus != PC_SUCCESS)
   {
      printf(“Proxy creation has failed\n”);
      return 0;
   }


//Submit the subscription.
   nStatus = PC_evn_subscribe_with_exp(
```

```
        p,"Sample.Chat",
        "UserName==\"Jane\"UserName==\"Jane\"",
        notify1,
        strdup("Listen to Jane"),
        PC_NULL, &hSubscription1);
    if (nStatus != PC_SUCCESS)
    {
        printf("Subscription submission has
failed\n");
        return 0;
    }

    nStatus = PC_evn_subscribe_with_exp(
        p,"Sample.Chat",
        "((Gender == 'F' or Gender == 'f')
        and Age < 14)",
    notify1,strdup("Girl Scouts"),
    PC_NULL, &hSubscription2);

    if (nStatus != PC_SUCCESS)
    {
        printf("Subscription submission has failed
            \n");
        return 0;
    }

    nStatus = PC_evn_subscribe_with_exp(
        p,"Sample.Chat",
        "Interest == \"Country Music\"Music\"",
        notify2,strdup("Country music fans"),
        PC_NULL, &hSubscription3);
    if (nStatus != PC_SUCCESS)
    {
        printf("Subscription submission has failed
            \n");
        return 0;
    }
        printf("Subscription was submitted...
            waiting 30 seconds for incoming notifications
            \n");
        sleep(30);
}
```