Information and Content Exchange

ICE Implementation Cookbook

Getting Started with Web Syndication

From Members of the ICE Authoring Group

Adam Souzis, Laird Popkin, Sami Khoury, Bruce Hunt

This is a collection of recipes for implementing ICE in a step-by-step manner.

ICE is hosted by IDEAlliance , Solutions Through Standards
© 2000 IDEAlliance | 100 Daingerfield Road | Alexandria, VA 22314
703-519-8191 | www.IDEAlliance.org

ICE Implementation Cook Book

Acknowledgements

All of the ICE Authoring Group contributed to this Cookbook. In particular, we wish to thank Dianne Kennedy, Linda Burman, Michael Branch, Michael Strong and Nathan Pride for their comments on the drafts. The editors have shamelessly used all their good advice. The Ice Authoring Group is one of the IDEAlliance standards groups. This document was inspired by Brad Husick whose resemblance to Brad's Gadgets is purely coincidental. ;-) The authors have no idea why the names are the same. We further disclaim all knowledge of anyone named Joe Cool.

Unfortunately there are likely to be some typographical errors in this document. If you find one, please send an E-mail to any of the editors above. We also solicit your implementation experience and feedback on this step-by-step approach.

Use of this document may be enhanced by obtaining the latest ICE Specification. For the latest Information on ICE please see http://www.icestandard.net and consider joining the ICE network!

Preface

This ICE Cookbook was written by the authors of the ICE (Information & Content Exchange) specification to assist in implementing ICE in a wide variety of situations. Every system designer attempts to balance the near term need for an immediate solution with the longer term desire for a well structured and complete solution. The authors recognize the need for solving simple problems with simple solutions, and more complex problems with more sophisticated answers. The recipes included in this cookbook start with the simplest of all solutions and build on each other so that each succeeding recipe further adds to the resulting ICE implementation. Although the resulting solutions are not 100% ICE compliant, this practical approach to building ICE functionality over time is well proven in the marketplace. The objective is start with the minimal working protocol using ICE constructs and add capability until a minimum ICE compliant implementation is achieved. This implementation cookbook is a way to provide the Web community with a set of practical implementation steps that both get the job done quickly and efficiently as well as providing a path so that your near term investment in building an easy solution assists the longer term goal of full inter-operability with commercial ICE implementations.

1. Introduction

The ICE specification consists of over one hundred pages describing the ICE protocol. When you read the specification it is difficult to know where to begin an ICE implementation. This cook book, on the other hand, is designed to give you a step-by-step set of recipes for implementing ICE. It starts with a minimum protocol subset that is not ICE compliant, but it works and will enable you to begin syndicating content. Each following recipe builds on the previous one to add additional capabilities until a minimal, conforming ICE implementation is achieved. Thus, this document provides a road map for implementing ICE.

1.1 Basic Terms

Although this cook book is designed to provide simple ICE implementation advice, some basic nomenclature is required to understand the text:

ICE:
Information and Content Exchange protocol
Syndicator:
A content aggregator and distributor.
Subscriber:
A content consumer.
Subscription:
An agreement between a subscriber and a Syndicator for the delivery of content according to the delivery policy and other parameters in the agreement.
Collection:
The current content of a subscription.
ICE Package:
A delivery instance of commands to update a collection such as the addition of content items.
ICE Payload:
The XML document used by ICE to carry protocol information. Examples include requests for packages, packages, catalogs of subscription offers, usage logs and other management information.

1.2 About the Recipes

The recipes are as follows:

1. **Pull public content**. You obtain the capability to pull or link to content using the simplest ICE package processing and the use of item metadata. While this recipe provides a set of simple services, it is not ICE compliant.
2. **Non-compliant minimal ICE spelling for content delivery**. With this recipe you obtain all of the facilities of recipe #1 plus access control, personalization of subscription content and you are on the road to complying with the full ICE specification.
3. **Nearly ICE compliant, minimal implementation**. This recipe enables you to obtain confirmed delivery after processing, notification and restricted-offer support.
4. **Full ICE compliance.** In addition to all of the above, this recipe enables you to obtain automated subscriptions and subscription management. You can also support trivial negotiation and you gain the benefit of ICE inter-operability.
5. **Enhancements:** You can enhance each recipe in a number of areas to achieve increased automation and efficiency in handling syndication relationships. We describe easy enhancements at each of the recipe steps.

1.3 Examples

To make the recipes concrete and complete, we'll provide an example. We'll highlight the places where you will replace example specific implementation information with your own changes so that you can easily see how to apply the recipe to your environment.

The example we'll use is the following.

We'll look at both sides of content supply and consumption to illustrate how syndication works.

Consider first that you are interested in news and views about the latest in personal electronic gadgets. Let's suppose that your name is "Joe Cool" and you want to regularly obtain the latest information about the wonderful world of emerging personal electronic gadgets. This makes you an ideal subscriber (a content consumer) if you find someone that provides this information.

Consider second that you want to provide others access to all the information about personal electronic equipment (gadgets!) that you've been avidly collecting. To do this, you might put up a web site; or you could become a content Syndicator (a content supplier). As a content Syndicator, you will provide daily updates of the latest news to interested subscribers. As a content Syndicator, let's suppose your name is "Brad" and your domain name is "BradsGadgets". You share your extraordinary contacts and insight into the gadget world through various content feeds on your web site, such as newsletters, reports, columns, etc. It is well known that Brad is interested in various gadget areas and has information and opinions on Digital Cameras, Personal Digital Assistants, Media Players, Cell Phones and Laptop Computers.

1.4 Recipe Organization

Now, a word about the recipe organization. Each of the recipes has a background section that sets up the problem that the recipe solves and describes the general form of the solution. This is followed by implementation guidelines for Syndicators and subscribers. These implementation guidelines are the real meat of the recipe. The guidelines are laid out in a step-by-step manner using parts of the example above to make it concrete. We've provided sample XML documents and sample URL references as well as descriptions of what is needed for code. Where we think you'll want to make changes to fit your application, we've highlighted the samples using bold-italics like this:

http://www.bradsgadgets.com/ice/subscribers/joecool.ice

When you see a phrase in bold italics, you should think about how you want to replace it for your application.

After the implementation guidelines, we include a section on enhancements that you can make to the recipe to achieve increased automation, function or inter-operability while building toward a fully compliant well featured implementation. Finally, we have a section on assumptions based on the ICE specification as well as how the recipe conforms to the ICE specification. We do this so you'll know where this recipe is at odds with a full ICE specification and where we've taken advantage of the flexibility in the ICE specification to achieve this implementation model. This will be handy as you plan how you want to make extensions or improvements to the recipe when you build your own implementation.

Maximum inter-operability is achieved when everyone you communicate with knows how to respond to your protocol. The section on conformance is designed to assist you in planning upgrades to maximize the number of other Syndicators (and subscribers) that you can communicate with. This enhances the value of your implementation and lowers your syndication and/or subscription costs.

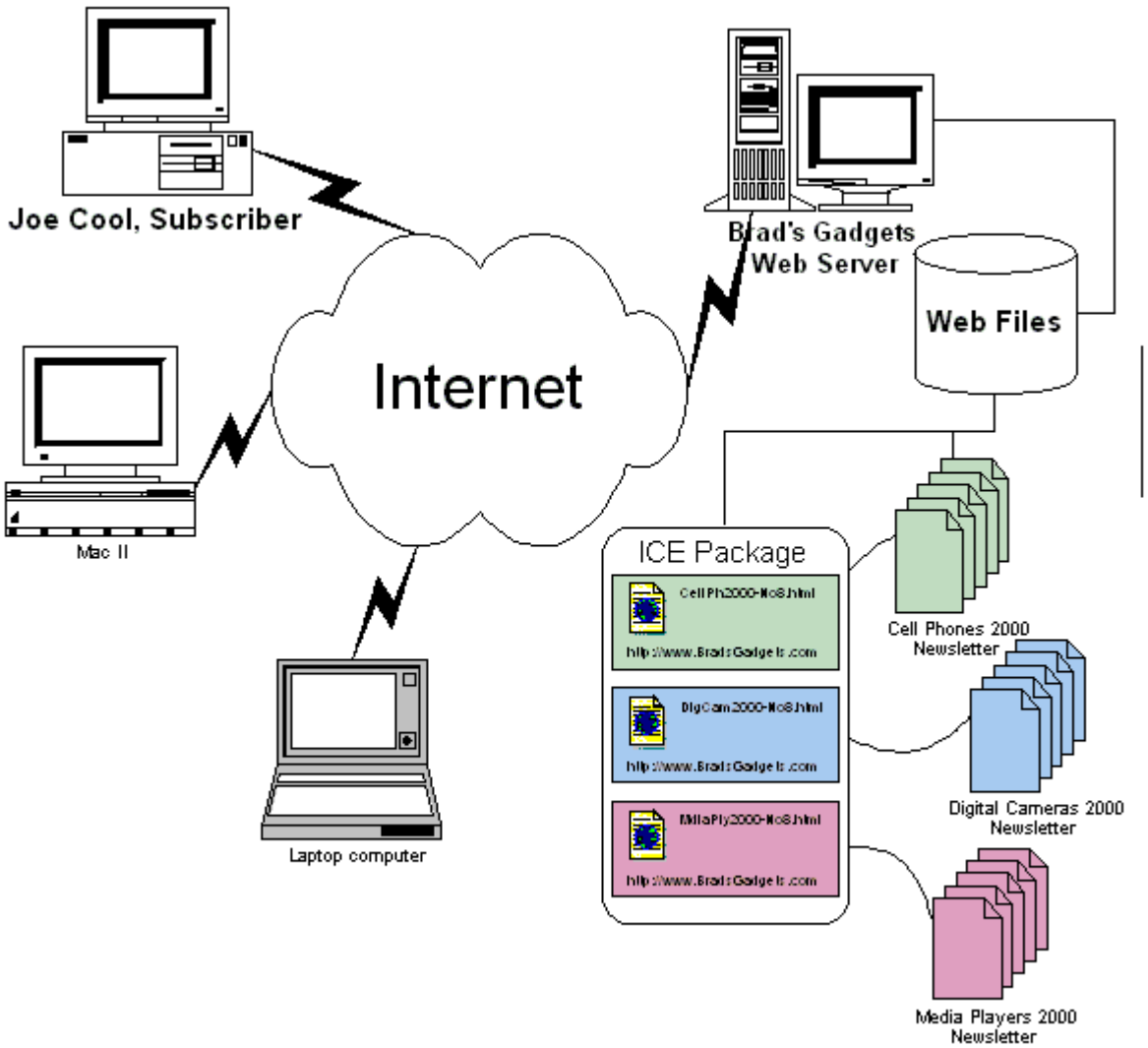
Let's get to it. We certainly don't want the ICE to melt. (Bad ICE jokes are strictly encouraged!)

2. Recipes

2.1 Pull Public Content.

2.1.1 Background

In this recipe, a Syndicator provides content by generating two sets of publicly accessible files on its HTTP server. One set of files contains the media content and the other set of files are ICE packages. A subscriber accesses a content offerings by retrieving an ICE package and processing it. The Syndicator, creates an ICE package for each offering of content it wants to provide. The following picture illustrates the setup:



Each ICE package contains one or more pointers to the content as well as providing metadata that describes how the content can be used and with simple enhancements, how long the content will be available as well as a modicum of content protection. This allows the subscriber to know when to check back for updates for when the content will be removed. The approach provides a number of advantages to Syndicators and subscribers. For subscribers, they know the lifetime of the content so that its use can be appropriately planned. Subscribers also know when to check for updates. Syndicators can change content and update content without disturbing existing subscribers. This way both Syndicators and subscribers avoid "404" not found errors. Finally, syndicators can notify subscribers of limitations on use of the content using ICE item metadata.

2.1.2 Syndicator Implementation Steps

1. Construct the ***newsletter***, "***Tech Tips from Brads Gadgets***" using your normal production process. We'll assume that you place it on your site under the following URL:
   http://www.bradsgadgets.com/news/techtips/<todaysdate>/news.html
2. Construct a standard ice-package:

```
<?xml version="1.0"?>
<!DOCTYPE ice-package SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd">
<ice-package>
   <ice-item-ref
          url="http://www.bradsgadgets.com/news/techtips/2000-08-01/news.html"
          item-id="BGTECHTIPS_V1" />
</ice-package>
```

   The item-id uniquely identifies the content within the package.

3. Place the ice-package at some public location on your web-site, say ***http://www.bradsgadgets.com/ice/techtips.ice***

That's all that is required of the Syndicator. Note that the Syndicator may wish to publicize the standards location for public syndications using ICE.

2.1.3 Subscriber Implementation Guidelines

1. Obtain the URL for the simple ICE based content, the ice-package, from a potential Syndicator. In this example, you obtain the URL:

   http://www.bradsgadgets.com/ice/techtips.ice

   Periodically you will refetch the URL to update to the next newsletter.

2. Parse the ice-item-ref URLs out of the ice-package and either download the content or reference it using the URL

That's all there is to it! You now have the simplest possible syndication.

2.1.4 Enhancements

2.1.4.1 Multiple URLs in a single package

You can repeat the ice-item-ref in a package as many times as you like to provide many different pieces of content. Suppose for example, that Brad wants to have a newsletter for MP3 players and for Digital Cameras. He would then construct an ICE package like the following:

```
<?xml version="1.0"?>
<!DOCTYPE ice-package SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-package >
   <ice-item-ref
          url="http://www.bradsgadgets.com/news/techtips/2000-08-01/MP3news.html"
          item-id="BGMP3TIPS_V1" />
   <ice-item-ref
          url="http://www.bradsgadgets.com/news/techtips/2000-07-01/DCnews.html"
          item-id="BGDIGCAMTIPS_V1" />
</ice-package >
```

The advantage here of course is multiple documents in a single easy to access package. Notice that the subscriber does exactly the same actions as before except now there are two newsletters.

2.1.4.2 Simple content management using Access Control

While the above recipe allows you to syndicate simply and very quickly, it has a number of simple improvements that can give you significant benefit. There are two simple improvements that you can do to manage your content effectively. The first is to notify your subscribers how long the content is available. This allows you to control the lifetime of your content and it means that your subscribers can plan for it's effective use without encountering "page not found" 404 errors. The second is to control who can access your content by using HTTP's basic authentication (password protection) mechanism.

Setting the length of time your content can be accessed.

An ice-item-ref has an access control mechanism. You can quickly enhance this recipe to notify your subscribers of the lifetime of the content -- i.e. when it will be available and for how long. So, now the content of the ice-item-ref element (which was EMPTY) gets replaced with:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-package SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-package >
   <ice-item-ref
      url="http://www.bradsgadgets.com/news/techtips/<todaysdate>/news.html"
          item-id="BG1" >
      <ice-access >
         <ice-access-window starttime="2000-07-21T08:00:00"
                            stoptime="2001-08-01T00:00:00" />
      </ice-access >
   </ice-item-ref >
</ice-package >
```

The ice access window element specifies the span of time that the URL will be available. This means that you as a subscriber now know how long the link will last. And, you as the Syndicator have made a commitment to provide it for a specific length of time. Thus, the duration of the content is now made explicit so that each party can avoid the "404" broken link problem. This simple addition can save your subscribers many hours of wondering if a "404" is temporary or permanent. Also, notice that you can update the access window if you decide to extend or reduce the lifetime of the content. Thus, your subscribers only need to refetch the `ice-package` to find out the current lifetime *before* they call you!

Protecting Content using HTTP basic authentication.

The ice access control mechanism permits a Syndicator to provide simple protection for the content. This informs subscribers that HTTP's basic authentication mechanism protects the content behind the URL. You can enhance this protection by sending the `ice-package` through a private channel to prospective subscribers. For example, Brad sends this package by E-mail to his subscribers:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-package SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-package >
   <ice-item-ref
      url="http://www.bradsgadgets.com/news/techtips/2000-08-01/news.html"
          item-id="BG1" >
      <ice-access >
         <ice-access-window starttime="2000-08-01T00:00:00"
                            stoptime="2001-08-31T24:00:00" />
         <ice-access-control control-type="password"
                             user="TechTips Subscriber"
                             password="mysecret" >
           (C)2000 Brads Gadgets, Inc. All Rights Reserved.
           Use of the content in this item reference
           implies acceptance of the use license at
           http://www.bradsgadgets.com/licenses/subscriber.html
           including honoring all copyrights and trademarks.
           You agree not to provide others with the
           access control password above.
         </ice-access-control >
      </ice-access >
   </ice-item-ref >
</ice-package >
```

When Joe Cool receives this package, he is informed not only of the lifetime of the content, but he also knows that access to the content will require satisfaction of HTTP's basic authentication mechanism using the user and password provided. Only parties with access to the above package are able to get access to the content. The security of this facility is enhanced by delivering it over a private channel to prospective subscribers instead of

simply placing it on Brads web site.

Clearly this marks the content as being licensed. For you as a subscriber, it lets you know the fair use of the content because the usage rules are made explicit. This mechanism provides a defense against unfounded accusations of misappropriation of intellectual property including copyright infringement. For you as a Syndicator, it provides you the means to both deliver content and clearly mark the usage rules. Note that the access control mechanism does not prevent syndication of any specific type of content (public or private). It only provides a means to make the usage rules explicit and it provides a means to limit the population of users.

2.2 Request-Response Protocol - ICE spelling for content delivery.

2.2.1 Background

This recipe introduces ICE's basic request/response protocol to enable personalized content delivery for each of your subscribers.

Build a simple request/response protocol on top of HTTP.

In the previous recipe no programming was required. This recipe requires that the rudiments of a protocol be built. ICE uses a request/response protocol. In this protocol, every request is answered by a response. The requester repeatedly sends the request until it hears the response or it gives up, being fairly certain that the receiver is incapable of responding. The receiver may respond to the requester multiple times for a single request.

ICE relies on a reliable transport protocol such as HTTP. In this cookbook we assume that you'll use HTTP. The key ICE transport requirement is that the transport protocol deliver each `ice-payload` without error from the sender to receiver; or fail to deliver the `ice-payload` and report an error to the sender. This means that if the transport hands an `ice-payload` to the receiver, it has no errors. HTTP satisfies this requirement if you map `ice-requests` to HTTP Posts and map `ice-responses` to the HTTP response to an HTTP Post. Both sides (Syndicator and Subscriber) must be able to issue HTTP Posts and responses to HTTP Posts.

You can use the code that you developed in recipe one above to analyze and respond to the ice-package. In this recipe, however, you'll deal with full ice-payloads as well as ice-packages.

You get the following benefit from this recipe:

The syndicator will be able to:

- Provide multiple content collections each capable of being subscribed to*.
- Let multiple subscribers subscribe to one content collection*.
- Let one subscriber subscribe to multiple content collections*.
- Select the population that has access to syndicator content.
- Control how long the content is available
- Directly deliver a simple ice-package to each subscriber.
- Lay foundation for automated delivery

(*-capability available in Recipe 1)

The subscriber will be able to:

- Have multiple subscriptions
- subscribe to content from different Syndicators
- subscribe to different collections from the same Syndicator
- potentially gain access to more exclusive information

Syndication Scenario - Brad has multiple newsletters to syndicate.

Suppose that Brads initial foray into syndication has been extremely popular. So, Brad begins selling his gadget reviews to subscribers and upgrades his offerings to send content differentiated reviews to different subscriber populations. Brad writes newsletters on Digital Cameras, Laptop Computers, Cell Phones and Media Players.

Brad couples his Web site to an ICE site. The ICE site allows him to offer multiple different newsletters as subscriptions and to keep a list of subscribers for each of the newsletters. Also, the ICE site sends a newsletter to any of the subscribers when they request it. Brad uses his Web site to solicit customers for his newsletters and to provide an easy, interactive way to initiate subscriptions.

Joe Cool, for his part wants to obtain only the newsletters about Digital Cameras and Cell Phones and is willing to pay a small subscription fee.

In this recipe, the subscriber, (Joe Cool), is willing to retrieve the newsletters for which he has a subscription every month.

2.2.2 Syndication Implementation Guidelines

1. A new subscriber, Joe Cool, signs up to receive content from Brad's Gadgets. How does Joe Cool know how to sign up? Brad solicits new subscribers on his web site and creates a forms based subscription application. Joe Cool while browsing the site, finds the application and fills out the form. Brad creates a new subscriber account from the subscription application using his Syndicator software and assigns a subscriber identifier for Joe. He then sends Joe Cool his subscriber id, say, ***4af37b30-2c35-11d2-be4a-204c4f4f5020***; and also a subscription id, say, ***SB-BradsGadgets-DigitalCameras2000-2000-08-23S3-003F9A7C*** for the Digital Camera newsletter and, say, ***SB-BradsGadgets-CellPhones2000-2000-08-23S14-003F9A7C*** for the Cell Phones newsletter. To retrieve the content , the subscriber, Joe Cool, also needs to know the URL Brads syndicator is running on so that he can post requests there. Brad sets up the following URL: ***http://www.bradsgadgets.com/ice/newsletters***.

   A Few words about identifiers. If you are going to implement a syndicator, you will need to generate several identifiers as indicated above and below. Most of these identifiers need to be unique within your syndication relationships but that is about all. However, each syndicator and each subscriber needs a universally unique identifier to prevent duplication. ICE uses the UUID (see Universally Unique Identifiers) as a distributed means to provide unique names for subscribers and syndicators. Once you select a UUID for your syndicator or subscriber, you should use it from then on. The UUID is how you as a subscriber know for sure which syndicator you are communicating with and similarly for you as a syndicator, this is how you uniquely know which subscriber it is. The style and format of the other identifiers is up to you within the uniqueness constraints. Below in Appendix B are models for various identifiers.

2. Brad prepares the newsletters in his usual way. When they are ready to "publish", Brad copies them to a place on his Web site that is accessible and known to the syndication software. (This location may or may not be publicly accessible; but it must be accessible to the syndication software.). The syndicator software is now ready to provide content to subscribers.

3. The syndicator, Brad, receives an ICE payload containing an ice-get-package request posted to the Syndicators URL by the subscriber, Joe Cool's, ICE client software. This happened because Joe Cool monitors Brads Web site to find out when the new issue of the newsletters are available. Brad puts up a page informing subscribers and potential subscribers that the newsletters are available. Joe Cool then turns on his simple subscriber client and sends a request to Brads simple ICE syndicator. For example Joe Cool sends the following request:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:02:23.007-JoeCool-39"
     timestamp="02:02:23,449" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>IceBlock Systems ICE Processor, V7.0</ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-07-21T02:02:23_JoeCool_58" >
      <ice-get-package current-state="ICE-ANY"
        subscription-id="SB-BradsGadgets-DigitalCameras2000-2000-08-23S3-003F9A7C" />
   </ice-request>
</ice-payload>
```

The `payload-id` uniquely identifies the payload for the sender (in this case, the subscriber, Joe Cool). The `request-id`, set by the sender will be used by the receiver (in this case, the syndicator, Brad) in his response so that the sender knows exactly which request was meant in the response. Of course, the `subscription-id` tells Brad which newsletter is desired. With this complete, notice that Brad has enough information to look up the subscriber and the right subscription. Brad can also check to make sure that Joe Cool is really a subscriber; and so can precisely respond to the request.

A similar payload is easy to generate for the Cell Phone newsletter. Notice that the identifiers are chosen for communication clarity. You may want to use simpler ones; or even choose a format that is difficult to predict. The latter are an aid in preventing undesired access to the content. See Appendix B for some sample identifier prototypes.

4. The syndicator, Brad, looks up the sender to see if the requester is a valid subscriber. If not, an unknown subscriber error message is returned (See Appendix A: Unknown Subscriber Error Response for the format.) If valid, the syndicator responds with an `ice-payload` containing a personalized `ice-package`. In this case, Brad responds with the following payload:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:02:24.019-BradsGadgets-1327"
     timestamp="02:02:24,016314"
     location="//www.bradsgadget.com/ice/newsletters"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
         RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-1" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-07-21T02:02:23_JoeCool_58" />
      <ice-package new-state="ICE-ANY" old-state="ICE-ANY" package-id="CP20000701-1">
         <ice-item-ref             url="http://www.bradsgadgets.com/ice/newsletters/D
              item-id="BG_JoeCool_1" />
      </ice-package>
   </ice-response>
```

```
</ice-payload>
```

The sender-id here is Brads UUID. The response for the Cell Phone newsletter is similar in pattern to the above. Again, the identifiers are chosen for reader clarity. You may wish to obscure them with a format that is more difficult to predict.

The `message-id` in the `ice-code` is identical to the `request-id` from a previous request. This way the receiver knows that this response is for a previously sent request and which one it was.

5. Any other requests are responded to with "not implemented" (503) ice-code such as that shown in Appendix A.3. That's it. You know have a simple syndication capability.

2.2.3 Subscriber Implementation Guidelines

1. The subscriber, Joe Cool, wants to subscribe to content being offered from Brads Gadgets. He contacts Brad to setup the subscription. In this example, this is done by Joe perusing Brads Web site and deciding to fill out the HTML forms to subscribe to the Digital Camera and Cell Phone newsletters.
2. Brad sends Joe a subscription ID,
   "***SB-BradsGadgets-DigitalCameras2000-2000-08-23S3-003F9A7C",***
   for the Digital Camera newsletter and
   "SB-***BradsGadgets-CellPhones2000-2000-08-23S14-003F9A7C***",
   for the Cell Phone newsletter as well as the URL pointing to Brads ICE syndication software,
   http://www.bradsgadgets.com/ice/newsletters.
   Joe obtains a UUID for his subscriber, say, "***4af37b30-2c35-11d2-be4a-204c4f4f5020" .***
3. Joe enters this information into his ICE client software and points at Brads site using the URL Brad supplied. Joe then peruses Brads site looking for the availability of the newsletters he subscribes to. Note that in this recipe, Joe manually tells the simple subscriber software to attempt to retrieve the newsletters.
4. The subscriber ICE client software sends (via an HTTP post) an ICE payload containing an `ice-get-package` request that is similar to that shown in step 3 of 2.2.2 above. For example, the simple subscriber processor might send the following to obtain the Cell Phone newsletter:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:02:24.019-JoeCool-14"
     timestamp="02:02:23,449" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-07-21T02:02:23-JoeCool-1" >
      <ice-get-package
        current-state="ICE-ANY"
   subscription-id="SB-BradsGadgets-CellPhones2000-2000-08-23S14-003F9A7C" />
   </ice-request>
</ice-payload>
```

5. The ICE client software receives the ice-package. In this example, it would look like:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:03:45.019-BradsGadgets-1523"
     timestamp="02:03:45,314"
     location="//www.bradsgadget.com/ice/newsletters"
     ice.version="1.1" >
   <ice-header >
```

```
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-1234" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-07-21T02:02:23-JoeCool-1" />
      <ice-package new-state="ICE-ANY" old-state="ICE-ANY" package-id="CP20000701-1">
         <ice-item-ref            url="http://www.bradsgadgets.com/ice/newsletters/C
              item-id="BG_JoeCool_1" />
      </ice-package>
   </ice-response>
</ice-payload>
```

The ICE client software processes it as in recipe 1 above. That's all there is to it.

In your implementation you may want to have timers associated with the requests that you send out. If the timer expires before you get a response, you can send the request again. If you count the number of times this happens, when it exceeds some number that you select, you can be fairly sure that the Syndicator is having problems; or that you are posting to the wrong URL. When this happens, it is probably time to check your URL and call the Syndicator for assistance.

2.2.4 Enhancements

There are many enhancements that are possible once this basic recipe is working. These enhancements improve the operation of content syndication in a number of ways including improving function, efficiency and error recovery.

2.2.4.1 Multiple URLs in a package and Access Control

Notice that the same enhancements for recipe 0 work here as well. That is, you can add access controls to set the availability time of the content as well as basic HTTP authentication access control.

2.2.4.2 Use other protocols to access Content.

The `ice-item-ref` elements can have the URL specifiy another protocol such as FTP or RTSP in the URL. Thus, you can direct your subscribers to the desired protocol to use to access the content.

2.2.4.3 Send Content In line as well as by Reference.

The Syndicator could send ice-items directly (i.e. in line) instead of an using an `ice-item-ref`.. If this is done, the subscriber must be able to handle "escaped" character data as is required by XML (i.e. both character entities and also base 64 encoding). This is an additional requirement for the implementation. To do this, the Syndicator would send `ice-packages` that contain both `ice-items` and possibly `ice-item-refs`. The subscriber then extracts the `ice-items` and `ice-item-refs` from the `ice-packages` in an `ice-payload`. Both the Syndicator and the Subscriber must implement the ability to handle `ice-items`. The following is a sample `ice-payload` that contains both an `ice-item` and an `ice-item-ref`. In it, Brad uses an `ice-item` to deliver news flashes that keep his subscribers up-to-date:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload
          SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:02:25.039-BradsGadgets-1631"
```

```
          timestamp="02:02:25,039416"
          location="//www.bradsgadget.com/ice/newsletters"
          ice.version="1.1" >
        <ice-header >
           <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                       name="Brads Gadgets, Inc." role="syndicator" />
           <ice-user-agent>
                 RockSolid Protocols ICE Processor, V17
           </ice-user-agent >
        </ice-header >
        <ice-response response-id="RSP-2000-07-21T02:03:45_BradsGadgets_1234" >
           <ice-code numeric="200" phrase="OK"
                     message-id="REQ-2000-07-21T02:02:23_JoeCool_1" />
           <ice-package new-state="ICE-ANY" old-state="ICE-ANY" package-id="CP20000701-1">
              <ice-item name="WMLstory"
                        subscription-element="NewsFlash"
                        item-id="NewsFlash1" >
                WML adopted by major cell phone providers including
                Motorola, Nokia, Qualcomm, Erricson, NEC, Samsung and
                Phillips.  In a stunning show of Internet inter-operability,
                the major manufacturers today announced that they
                will support WML for Internet access...
                .
                .
                .
              </ice-item>
              <ice-item-ref  url="http://www.bradsgadgets.com/ice/newsletters/CellPhone/Ju
                   item-id="BG_JoeCool_1" />
           </ice-package>
        </ice-response>
     </ice-payload>
```

As you can see, the ability to directly deliver content can enhance offerings. The part that you need to be careful about is the case where the ice-item contains HTML or other marked up text. There are two ways of handling this.

The first way to handle "markup" is to use a CDATA section. Place the HTML or other markup content in the interior of "<![CDATA[" and "]]>". For example, suppose that you wanted to send the following HTML fragment in line:

```
<![CDATA[
<html>
  <head>
    <meta http-equiv="content-type"
          content="text/html;charset=iso-8859-1">
    <meta name="generator" content="Adobe GoLive 4">
    <title>Cell Phones 2000 - Hot August Nights Issue</title>
 </head>
 <body bgcolor="#e5daa9">
   <h1>Cell Phones 2000 - Hot August Nights Issue</h1>
   <h2>(C)2000 Brads Gadgets, Inc. All rights reserved!</h2>
   <p>Well phone phreaks, while we've been sweltering in the
      hot August sun, the phone phaetons have been busy pouring
      plastic and pushing photons to create ourcerebrall cellular means
      of communication.  In this phreaky issue we look at the
      latest cellular cool from Phillips.  While these guys have
      been pretty sedate as of late, now they are phreaking!
      ...
   </p>
   ...
  </body>
</html>
]]>
```

This content would be placed in an `ice-item` in an `ice-package`. The CDATA section cannot nest - so you put only one CDATA wrapper in for each item. This mechanism works pretty well until you start working with paired right box brackets followed by a right angle bracket ("]]>").

The second way to handle mark-up is to encode the markup using the standard XML escapes:

- > for ">" (right angle bracket).
- < for "<" (left angle bracket).
- ' for an apostrophe (').
- & for an ampersand, "&".
- " for a quotation mark (").

This approach can get fairly tedious and it expands the size of the file; but this approach lets you send any sequence of text including the paired right box brackets followed by a right angle bracket ("]]>").

2.2.4.4 Implement a "Ping" facility.

One of the most useful facilities for any protocol is the ability to "ping" the other side and get a response back. Receipt of a response quickly isolates drives problem resolution toward the ICE like implementation or to making sure that the transport mechanism is working. To implement a ping facility all that is needed is to implement the ice-nop request and a very simple response.

A model for the `ice-nop` request is:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:02:25.027-JoeCool-348"
     timestamp="02:02:25,027341" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-07-21T02:02:23-JoeCool-2397" >
      <ice-nop />
   </ice-request>
</ice-payload>
```

A model for the `ice-nop` response is:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:03:45.132-BradsGadgets-1787"
     timestamp="02:03:45,31416"
     location="www.bradsgadget.com/ice/newsletters"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-1234" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-07-21T02:02:23-JoeCool-2397" />
```

```
    </ice-response>
</ice-payload>
```

Both the Syndicator and the Subscriber should implement this request and response. Now subscribers and syndicators can "ping" each others server to assure that the communication channel is operative.

2.3 Nearly ICE compliant, minimal implementation

2.3.1 Background

This recipe expands the Syndicators ability to support many Subscribers. It also expands the ability of Subscribers to access many different Syndicators so that a wide selection of content can be obtained.

Enhance the request/response protocol with operational features.

This recipe enhances the previous recipe with many of the extras that make the ICE protocol particularly useful. Now that you have implemented the basic ICE request/response protocol it is time to add several useful operational features. Let's look at the features and their value:

Features and their Value to Syndicator and Subscriber:

Confirmed delivery processing
The subscriber notifies the syndicator when it has completed processing packages. The value here is that the both the Syndicator and Subscriber know that the content has been correctly processed.
Notification
Both parties can send operational notices to each other. The value here is that important operational information such as planned or emergency server outages can be communicated.
Ping facility
Both parties can send the simplest form of a request and response to assure each other that the communication channel is working properly. Note that this was an extension for recipe 2.2 above.
Fixed offers - no negotiation
The Syndicator automates the listing of offers and the Subscriber can pick up a list of offers from the Syndicator and choose the one it prefers.

Syndication Scenario

The popularity of Brads Gadgets has grown to the point that he is seeking a way to lower his operational costs so that his staff can get back to following the rapidly growing world of electronic gadgets instead of spending all their time setting up and managing subscriptions. He can do this in several ways. First, Brads Gadgets is fielding many support calls for troubleshooting subscriptions that are expensive and slow to resolve. It is especially frustrating when the problem is not in the syndication or subscription processors at all, but in the communication channel. A simple fast way of verifying the communication channel is needed. Many of the support calls arise when an emergency and/or routine maintenance require the servers be taken down. A way is needed to notify his subscribers of operational events.

Brads Gadgets has been approached by his customers with the desire to re-syndicate several of his newsletters. Brad wants to be sure that his new syndicating subscribers provide as high a quality of service as he provides and so wants to know that his content has been processed by his new syndicating subscribers.

2.3.2 Syndication Implementation Guidelines

1. Implement the basic Request-Response Syndicator side of the protocol as in 2.2 above. We use recipe 2.2 as the basis for this recipe.
2. Implement in line package content delivery. This was an enhancement for Recipe 2.2. If you implemented it there, then you don't have to do anything. Otherwise, implement a mechanism to partition your content into packages of ice-items (and/or) ice-item groups; as well as the simpler ice-item-ref. See the extension, 2.2.4.3 above.
3. Implement the `confirmation` attribute on the ice-package. This attribute has a value of "true" or "false". By default the value is "false"; so if the attribute is omitted, the action taken is as if the value is "false" - namely the subscriber is not expected to confirm package processing with an ice-code. It is up to you as a Syndicator when to ask for package confirmation. One reasonable policy is to ask for confirmation on a collection or subscription basis. On some of your collections, the value may be sufficiently high that you'd like to know that the subscriber received and processed the content. For example, suppose that Joe Cool wants to re syndicate Brads newsletters and Brad wants to make sure that Joe is processing his newsletters in a timely fashion. (Brads Gadgets has become so popular that many of his subscribers are re-syndicating his newsletters.) Thus, when the ice-package is sent to Joe Cool, the `confirmation="true"` attribute is placed on the ice-package.
4. Implement the Notification Request. This Request allows your syndication server to send operator notifications to your subscribers. The model for this request is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T04:02:24.539-BradsGadgets-2132"
     timestamp="04:02:23,449" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-07-21T02:02:23-BradsGadgets-9723" >
      <ice-notify priority="5" >
        <ice-text>
           The special Cell Phones 2000 Consumer
           Electronics Show edition will be available
           starting in mid March when the show opens
           and will be updated daily during the run
           of the show.  You may want to re-fetch
           this issue multiple times to keep up to
           date with the latest news from CES.  The
           show runs from March 15 to March 19
           this year.  Thanks for keeping up with
           all your favorite Gadgets with the number
           one provider of Gadget Gossip.
           Thanks for your continued patronage,
           The ICE team at BradsGadgets.
        </ice-text>
      </ice-notify>
   </ice-request>
</ice-payload>
```

   The priority attribute value ranges from 1 to 5 and is required. Priority 1 is the most important; while priority 5 is the least important. A priority 1 message should cause appropriate emergency action at the receivers site. You should use this when an extremely important message needs to be delivered and acted upon by the operations management. You should also work very hard to not overuse this facility.

5. Implement the response to a notification request. You should display the notification on the server console (or computer screen). If the priority is a 1 or 2 you may want to consider using a dialog box and/or appropriate action to gain attention (Perhaps blinking text with a bell?). The model for responding to the

request is:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:04:23.699-BradsGadgets-2311
     timestamp="02:03:45,31416"
     location="www.BradsGadgets.com"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-9876" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-07-21T02:02:23-BradsGadgets-2397" />
   </ice-response>
</ice-payload>
```

If you implemented the `ice-nop` extension in the previous recipe, this step is very similar and should be even easier to implement. On the other hand, if you haven't implemented the `ice-nop`, now would be a great time to do so since you have most of it after completing this step; in fact we'll do it in the next step. The response to the notify request is just a straight-forward acknowledgement of the request.

6. Implement the `ice-nop`. See extension, 2.2.4.4 above for both the request and response side of the protocol.

When you have completed this recipe, you will have a robust, nearly ICE compliant, Syndicator. At this point, you've got a syndication processor that can handle many subscribers.

2.3.3 Subscriber Implementation Guidelines

1. Implement the Subscriber facilities to Recipe 2.2 above. As usual, we'll use the previous recipe as a starting point for this recipe.
2. Implement in line package content delivery. This was an enhancement for Recipe 2.2. If you implemented it there, then you don't have to do anything. Otherwise, implement a mechanism to partition your content into packages of ice-items (and/or) ice-item groups; as well as the simpler ice-item-ref. See the extension, 2.2.4.3 above.
3. Implement the ability to "confirm processing" within in your Subscriber processor. When a package is delivered, a new attribute, `confirmation,` may appear on the ice-package element. It asks you to send back a confirmation request to the sender of the ice-package when you have completed processing the ice-package. This allows you to inform the sender of your success at obtaining and understanding of the content in the package. It is generally good form to confirm when you've obtained all of the items in a packages without error and have updated your copy of the Syndicators collection. You, of course, may delay confirmation until you've actually used the delivered content. Upon correctly processing the items in a package to update your copy of the collection you send a request similar to the one that Joe Cool sends (in your standard payload) back to the Syndicator:

```
<ice-request request-id="REQ-2000-07-21T02:05:32-JoeCool-1873" >
   <ice-code numeric="201" phrase="Confirmed"
             message-id="REQ-2000-07-21T02:02:23-BradsGadgets-2397" />
</ice-request>
```

If processing failed for, say, inability to fetch the data in an ice-item-ref you provide a more sophisticated message:

```
<ice-request request-id="REQ-2000-07-21T02:03:00-JoeCool-1873" >
   <ice-code numeric="331"
             phrase="Failure fetching external data"
             message-id="REQ-2000-07-21T02:02:23-BradsGadgets-2397" >
     Unable to obtain content from URL:
     http://www.bradsgadgets.com/ice/newsletters/CellPhone/July2000.htm
   </ice-code >
</ice-request>
```

There are several ICE codes that can be used to describe to the package sender exactly what type of error in processing was encountered. Appendix A.4 contains a complete list of ICE codes that you can use. The following is a short list of those that you may want to use to announce non-confirmation:

- 201 - Confirmed. - The subscriber has successfully processed the package.
- 331 - Failure fetching external data
- 401 - Incomplete/Cannot parse - You couldn't even get the parser started!
- 402 - Not well formed XML - Your XML tags didn't balance.
- 403 - Request/ResponseValidation Failure - XML didn't match the schema.
- 405 - Unrecognized Sender - Who are you?
- 406 - Unrecognized Subscription - You have to have a valid subscription-id
- 407 - Unrecognized operation - An operation in the package was not one that you recognized
- 408 - Unrecognized operation arguments - The attributes on the element were unknown.
- 430 - Not Confirmed - (Generic error indicating that Subscriber didn't complete processing.
- 501 - Temporary Responder error. - I'm to busy right now to do it" message
- 603 - No more confirmations to send. - When you've confirmed everything, respond with this.

There are many others that are defined in the ICE specification. Since this is a request, you will receive an ice-code response and your standard retry mechanism (e.g. wait, time out & resend) should be used until you do.

Well, that's it for the confirmation feature.

4. Implement the Notification Request. This Request allows your subscription processor to send operator notifications to your syndicators. The model and discussion for this request is identical to that shown above for the Syndicator and is not repeated here.
5. Implement the `ice-nop`. See the Extension, 2.2.4.4 above for both the request and response side of the protocol.

   Well, that's it for this Recipe. Congratulations, these features should make it much easier to operate a subscriber.

2.3.4 Enhancements

2.3.4.1 Implement a simple scheduling system.

As a Syndicator, you may consider implementing a simple scheduling system. In Brads case, this system would periodically check for new newsletters in a specific place and if found, run through the list of subscribers to that newsletter creating notifications for "pull" subscribers and ice-packages for "push" subscribers.

As a Subscriber, you may consider implementing a simple scheduling system as well. In Joe Cools case, this system wakes up periodically and issues an ice-get-package request to his Syndicators to see if new newsletters are available. The rate at which the requests go out is determined by discussion with the Syndicator. BradsGadgets newsletters are both bi-weekly and monthly, so Joe Cool checks BradsGadgets on the first through fifth and the 15th through the 20th of every month until he successfully gets a newsletter or the checking

window expires.

2.3.4.2 Implement Content Push.

As a Syndicator, you can offer your subscriptions in either "push" mode or "pull" mode; as well as in line or by reference. In push mode, the Syndicator automatically delivers ice-payloads containing ice-packages whenever there is new content. In pull mode, the Syndicator relies on the Subscriber to initiate content delivery with an ice-get-package request. If you offer this service, you will need to provide a way for your subscribers (and potential subscribers) to sign up for the service. The implementation of this feature is very in-expensive since you already have the apparatus to construct ice-payloads containing content packages. All you need now is to build the apparatus that notices when new content is available, use your existing apparatus to build the ice-payload(s) and run through each of your subscribers to the content sending each one that is signed up for "push" delivery the payloads.

As a Subscriber, you can automate more of your subscriptions by using "push" mode delivery from a Syndicator. In this mode, the Syndicator will automatically send you ice-payloads with ice-packages whenever it generates new content. Of course, you have to have a receiver always ready to accept such packages. In some syndication operations that is not feasible. But, if you have a server up and running almost all of the time, it can make subscriptions rather easier to manage. Given your current subscription apparatus, you can easily add this facility. The same mechanism that takes ice-packages apart can be used for this facility; and you already know how to decode an ice-payload containing packages.

2.3.4.3 Implement Collection Management.

Background.

In previous recipes we have relied on Subscribers to keep track of their collections and have mostly dealt with collections that can be accessed via reference. Suppose that Brad decides to offer his complete collected newsletters as a single subscription. In this case, Brad can construct an ice-package that contains an ice-item-ref for each subscription. But what if the subscriber wants to have the newsletters sent in line? If Brad puts each newsletter into an ice-item, in a large package, it may be that the result is to large to send via HTTP because many HTTP servers limit the size of a post or response to the post. The solution, of course, is to break the large package into several smaller ones. But then, what if one of the packages is lost; arrives in error, or worse, is not sent by accident. Or, suppose that the packages arrive out of order through delayed delivery. How will the receiver know how to re-assemble them in the correct order? As you have guessed by now, ICE provides collection management to make sure that all these problems cannot occur. Notice that none of these problems arise (directly) if your content can always fit into a single package.

Basic Idea

The idea behind collection management is very simple. The Syndicator marks every package with a special identifier (yes, another identifier!) called a package-sequence-identifier. Whenever the Syndicator sends an ice-package to a Subscriber, the Syndicator sends the old package-sequence-identifer he thinks the Subscriber has; as well as the new package-sequence-identifier for this package. The Subscriber keeps track of its current package-sequence-identifier and holds on to it until it has successfully updated its collection with the new package; at which point it switches to the new package-sequence-identifier. If the old package-sequence-identifier does not match the Subscribers current identifier, the Subscriber sends an error message back to the Syndicator with it's current package-sequence-identifier; at which point, the Syndicator can send the correct sequence of packages. Notice that most of the time, the Subscriber has only one package-sequence-identifier to keep for a collection. The Syndicator and Subscriber can send each other requests to know what the current package-sequence-identifier is for the other. Getting the package-sequence-identifier from a Subscriber tells the Syndicator exactly what state the Subscribers collection is in. Getting the package-sequence-identifier from the Syndicator tells the Subscriber whether or not its

collection is up-to-date and so indicates to the Subscriber whether or not is issue a ice-get-package request.

Implementation

The Syndicator needs to generate an package-sequence-identifier (PSID) for each package it sends. It keeps track of the Subscribers PSID in each subscription for the Subscriber. A sample model for the PSID is in Appendix B.5 below.

2.4 Full ICE compliance

2.4.1 Background

Brad determines that he can reduce his costs further by standardizing his subscription offers to reduce the initial costs of starting up a subscription since this is another place in the syndication service that is costing much valuable staff time. Brad automates the subscription offerings by creating a catalog of subscription offers where subscribers can access them programmatically. This allows Brad to quickly make new offers to better reflect his changing content offerings and automatically make it available to his Subscribers. In addition, Brad is looking forward to implementing offer negotiation which will allow him and his subscribers to optimize their subscription operations. For Brad this will means that he will be able to reduce peak server loading as well as delay having to invest in new hardware and operating costs. For Brad's subscribers it means that they will be able to get better operational parameters to suit their needs such as better delivery times, dates and frequencies. For example, a large subscriber may want Brad's newsletters delivered in the early morning so as not to impact other site operations.

Value to Syndicator and Subscriber

Automated Subscription Selection
: The subscriber asks for a listing of subscription offers called (surprise!) a catalog. The subscriber can then select an offer based not only on the collection, but several other parameters that the Syndicator places in the offer. These parameters can either be subscription operational parameters such as delivery windows, etc; or any other set of business parameters that the Syndicator wishes to offer.

Subscription management
: The subscriber can now select a subscription offer from many different offers, can ask for subscription status, can change the subscription and cancel it. The Syndicator responds to all of these and can instigate a change in subscription as well. This includes the following:

Automated Subscription Establishment.
: A subscriber sends a selected subscription offer (`ice-offer`) to the Syndicator. The Syndicator accepts or rejects the offer.

Cancel Subscription
: A Subscriber can cancel a subscription (that is terminate it).

Get Subscription Status
: A Subscriber can ask the Syndicator for the current state of its subscription. This response include contact information, the current delivery policy and current length of time and number of deliveries left among other detailed information about the subscription.

Features

`ice-get-catalog`
: Subscriber may request, Syndicator Must respond

Fixed subscription offers
: Subscriber selects subscription offer from catalog, Syndicator only accepts or rejects

`ice-cancel`
Subscriber may issue an `ice-cancel`, Syndicator must respond to it.

`ice-change-subscription`
Subscriber may issue an `ice-change-subscription`, Syndicator must respond to it.

`ice-get-status`
Subscriber may issue an `ice-get-status` request. The Syndicator must respond with an `ice-status` which describes, in detail, the subscription.

2.4.2 Syndication Implementation Guidelines

1. Implement the receiver side of the ice-get-catalog request. The Syndicator responds to the ice-get-catalog request with an ice-catalog response. A good model for the ice-get-catalog request looks like:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:33.901-JoeCool-423"
     timestamp="22:10:33,741" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="2000-08-24T22:10:33_RQ_JoeCool_1873" >
      <ice-get-catalog />
   </ice-request>
</ice-payload>
```

An ICE catalog contains contact information and prototype subscriptions called offers. The contact information is used to aid in human to human communication so that when your subscribers need to call, they can quickly get in touch with the right people. Offers contain all of the proposed operating parameters for specific subscription. This includes all of the basic things like which collection to subscribe to, the name of the product being offered, how often it will be offered and a number of other parameters that give your customers a great deal of control over the subscription. In this Recipe, we'll limit ourselves to the smallest number needed. You can implement additional parameters as their value becomes apparent.

Below is a good model for a typical catalog response:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:45-BradsGadgets-2761"
     timestamp="22:10:45,321"
     location="www.BradsGadgets.com"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-9876" >
      <ice-code numeric="200"
                phrase="OK"
                message-id="REQ-2000-07-21T02:02:23-JoeCool-345" />
   </ice-response>
   <ice-catalog name="Brad Newsletters"
```

```
                url="http://www.BradsGadgets.com/offers/newsletters.html" >
      <ice-contact name="Brads Gadgets">
         For information please contact
         Catalog Offers:    Sherry Seller, 650-555-1212
         Technical Support: Gary Geek,     650-555-1313
      </ice-contact>
      <ice-offer product-name="Digital Cameras 2000 Newsletter"
                offer-id="BradsGadgets-DIGCAM-2000-V1-R1"
                subscription-id="ICE-NEW-SUBSCRIPTION"
                expiration-date="2001-08-30"
                quantity="20" >
        <ice-delivery-policy stopdate="2001-08-30" >
          <ice-delivery-rule mode="pull"
                max-num-updates="720"
                min-num-updates="13"
                max-update-interval="P2678400S"
                min-update-interval="P43200S" />
          <!-- max-num-updates is two per day,
               min-num-updates is 13 per year,
               max-update-interval is 31 days,
               min-update-interval is 12 hours -->
        </ice-delivery-policy >
      </ice-offer>
      <ice-offer product-name="Cell Phones 2000 Newsletter"
                offer-id="BradsGadgets-CELLPHONES-2000-V1-R1"
                subscription-id="ICE-NEW-SUBSCRIPTION"
                expiration-date="2001-08-30"
                quantity="30" >
        <ice-delivery-policy stopdate="2001-08-30" >
          <ice-delivery-rule mode="pull"
                max-num-updates="720"
                min-num-updates="26"
                max-update-interval="P1382400S"
                min-update-interval="P43200S" />
          <!-- max-num-updates is two per day,
               min-num-updates is 26 per year,
               max-update-interval is 16 days,
               min-update-interval is 12 hours -->
        </ice-delivery-policy >
      </ice-offer>
   </ice-catalog>
</ice-payload>
```

The above is a catalog of Brads Newsletters. In the catalog there are two fixed offers, one for the Digital Camera Newsletter and one for the Cell Phone Newsletter. It is easy to add additional Newsletters for the other gadgets Brad is interested in. The offers are fixed because there are no `ice-negotiables` specified. This indicates an unwillingness on the part of the Syndicator to negotiate any of the operational parameters.

The `offer-id` is a unique identifier used to specify directly this offer. No other offer in the Syndicators many offers can have this id. The `subscription-id` is set to the ICE 1.1 required "ICE-NEW-SUBSCRIPTION" value. This is an indicator that this is an offer and not yet a subscription. The Syndicator changes this to a valid `subscription-id` when it returns an acceptable offer to a potential subscriber *that the subscriber has selected and proposed*.

The `expiration-date` defines the date when the offer will expire. No subscription with this expiration date lasts beyond this date. You should choose this date based on the length of time that you are willing to deliver content according to this offer. If no `expiration-date` is provided on the `ice-offer`, then the subscriptions from this offer are permitted to run indefinitely. In most circumstances, it is not a good idea to allow either subscriptions nor offers to have unbounded time limits and therefore run indefinitely. (You get out of this case using the `ice-cancel` request.)

The attribute, `quantity`, sets the maximum number of updates during the time span of the offer. This means that the Syndicator promises to alter the collection at most this many times during the time of the offer. In the example, for cell phones, this is 20. The monthly newsletter is offered 12 times and the special CES update issue is updated daily during the show for 5 more times. The remaining 3 times are for other specials that might occur during the year. Thus `quantity` gives Syndicators an opportunity to set Subscriber delivery expectations.

An `ice-delivery-policy` is a list of delivery rules for the content specified by this subscription. The `stopdate` on the `ice-delivery-rule` defines the date at which each subscription constructed from this offer will end. What is implied but only indirectly stated is that the offer begins at the time the offer is sent. In this example, the date is 24 August, 2000. Notice that the subscription stops one year from the current (`ice-package`) date.

An `ice-delivery-rule` defines the parameters for content delivery. This is a very important XML element because it is how ICE knows when and how often to attempt delivery. In this example, we're only using a small number of the capabilities of one delivery rule. For sophisticated syndications, ICE permits many delivery rules to be applied to a single Subscription.

Lets look at some of the attributes. The `mode` of the delivery can be either `push` or `pull`. Push mode means that the Syndicator will send content to a subscriber; while pull mode means that the Subscriber will ask for the content. Both newsletters above are set to pull mode. The `mode` attribute is the only required attribute on an `ice-delivery-rule`.

The max-num-updates attribute stands for "maximum number of updates" and is set to 720 which during the year the subscription runs is (nearly) twice per day. ICE duration times are always in seconds. The max-update-interval attribute specifies the maximum amount of time that updates will be made available. These are set to 31 days for the Digital Cameras newsletter (or 2678400 seconds) and 16 days for the Cell Phones newsletter (or 1382400 seconds). The min-update-interval specifies the minimum amount of time that updates will be made available. These times are set to 12 hours (***43200 seconds)*** for both newsletters. These are a small number of parameters in a single delivery rule to keep the example simple.

While the catalog setup is the most sophisticated to date, notice that it is a simple list of offers so that potential Subscribers can choose a subscription. Each offer is a prototype subscription that is the layout of how the Syndicator thinks the subscription could work for a Subscriber. The Syndicator can construct many different offers to provide its collections in many different ways.

2. Implement the response to an ice-offer request. Brad finishes up automating support of subscriptions by responding to ice-offer requests. Potential subscribers select an offer from Brads Catalog and send a request similar to the following that was sent in by our buddy, Joe Cool:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:33-JoeCool-534"
     timestamp="22:10:33,741" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-000-08-24T22:10:33-JoeCool-1888" >
      <ice-offer product-name="Cell Phones 2000 Newsletter"
                 offer-id="BradsGadgets-CELLPHONES-2000-V1-R1"
                 subscription-id="ICE-NEW-SUBSCRIPTION"
```

```
                expiration-date="2001-08-30"
                quantity="30" >
        <ice-delivery-policy stopdate="2001-08-30" >
          <ice-delivery-rule mode="pull"
                max-num-updates="720"
                min-num-updates="26"
                max-update-interval="P1382400S"
                min-update-interval="P43200S" />
          <!-- max-num-updates is two per day,
               min-num-updates is 26 per year,
               max-update-interval is 16 days,
               min-update-interval is 12 hours -->
        </ice-delivery-policy >
      </ice-offer>
   </ice-request>
</ice-payload>
```

Brads responsibility is to check out the offer to make sure that he is willing to support the subscription in the offer. Brad can validate each of the parameters and should do so. Specifically, Brad pulls the offer-id and uses it to lookup his version of the offer as a comparison to validate the rest of the newly arrived offer. Brad then makes a decision whether or not he wants to engage in a syndication relationship with the sender. Presumably, you as a Syndicator would have a general business relationship setup with the potential subscriber and this would help determine if you will let them subscribe. In this example, Brad knows Joe Cool and is delighted to have Joe extend his subscription. Brad constructs a new subscription-id (See Appendix B.2 for ways to do this) and sends back the following ice-subscription to Joe Cool:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T23:40:45-BradsGadgets-3197"
     timestamp="23:40:45,321"
     location="www.BradsGadgets.com"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-9876" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-07-21T02:02:23-BradsGadgets-2397" />
   </ice-response>
   <ice-subscription current-state="INITIAL_STATE">
      <ice-offer product-name="Digital Cameras 2000 Newsletter"
                offer-id="BradsGadgets-DIGCAM-2000-V1-R1"
                subscription-id="SB-BradsGadgets-DigitalCameras2000-2000-08-23S3-003F9A7C"
                expiration-date="2001-08-30"
                quantity="20" >
        <ice-delivery-policy stopdate="2001-08-30" >
          <ice-delivery-rule mode="pull"
                max-num-updates="720"
                min-num-updates="13"
                max-update-interval="P2678400S"
                min-update-interval="P43200S" />
          <!-- max-num-updates is two per day,
               min-num-updates is 13 per year,
               max-update-interval is 31 days,
               min-update-interval is 12 hours -->
        </ice-delivery-policy >
      </ice-offer>
   </ice-catalog>
```

```
</ice-payload>
```

Notice that Brad has filled in the subscription-id with a new value. This is how Joe Cool asks for specific content.

3. Implement ice-cancel so subscriptions can be terminated.

   A Subscriber will issue an ice-cancel to terminate a subscription. Suppose for example that Joe Cool finds that he can no longer keep up with Digital Cameras and decides to focus his energy on his true passion, Cell Phones. Joe sends Brad the following:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:33-JoeCool-1349"
     timestamp="22:10:33,741" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-08-24T23:47:33-JoeCool-1983" >
      <ice-cancel
        subscription-id="SB-BradsGadgets-DigitalCameras2000-2000-08-23S3-003F9A7C" />
   </ice-request>
</ice-payload>
```

   The key items to note are the subscription-id which tells Brad which subscription to cancel. Brad verifies the sender-id and the subscription-id before cancelling the subscription. Brad then sends back the following response:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T23:40:45-BradsGadgets-2171"
     timestamp="23:40:45,321"
     location="www.BradsGadgets.com"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-9876" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-08-24T23:47:33-JoeCool-1983" />
   </ice-response>
</ice-payload >
```

   Note that Brad could also send back any of a number of status codes shown in Appendix A.4. For example, the status code "405" indicates an unknown sender and "406" indicates the subscription is unknown. It is straightforward for your implementation to reply with anyone of several status codes depending on the outcome of processing the request.

4. Implement ice-change-subscription to adjust subscription parameters.

2.4.3 Subscription Implementation Guidelines

1. Implement the `ice-get-catalog` request. The Syndicator responds to the `ice-get-catalog` request with an `ice-catalog` response. Joe Cool, now can ask for a catalog. A good model for the `ice-get-catalog` request looks like:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:33-JoeCool-1869"
     timestamp="22:10:33,741" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-08-24T22:10:33-JoeCool-1873" >
      <ice-get-catalog />
   </ice-request>
</ice-payload>
```

The model for the response to this request is shown in Step 7 above in the Syndicator Implementation guidelines. When your subscriber processor receives the ICE catalog, it should select one of the offers. One way to do it is to display the offer names and product-ids to a person for selection. For this implementation, the Subscriber-processor may simply select by product name. Joe Cool can now automatically select a subscription offer, suppose that he once again selects the Cell Phone newsletter Once selected, a model for the offer in our example would look like:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:45-JoeCool-1870"
     timestamp="22:10:45,741" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-08-24T22:10:33-JoeCool-1888" >
      <ice-offer product-name="Cell Phones 2000 Newsletter"
                 offer-id="BradsGadgets-CELLPHONES-2000-V1-R1"
                 subscription-id="ICE-NEW-SUBSCRIPTION"
                 expiration-date="2001-08-30"
                 quantity="30" >
        <ice-delivery-policy stopdate="2001-08-30" >
          <ice-delivery-rule mode="pull"
                 max-num-updates="720"
                 min-num-updates="26"
                 max-update-interval="P1382400S"
                 min-update-interval="P43200S" />
          <!-- max-num-updates is two per day,
               min-num-updates is 26 per year,
               max-update-interval is 16 days,
               min-update-interval is 12 hours -->
        </ice-delivery-policy >
      </ice-offer>
   </ice-request>
</ice-payload>
```

The Syndicator, BradsGadgets, checks the offer and responds with a new subscription:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
```

```
<ice-payload payload-id="PL-2000-08-24T23:40:45-BradsGadgets-4329"
     timestamp="23:40:45,321"
     location="www.BradsGadgets.com"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-9876" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-07-21T02:02:23-BradsGadgets-2397" />
   </ice-response>
   <ice-subscription current-state="INITIAL_STATE">
      <ice-offer product-name="Cell Phones 2000 Newsletter"
                 offer-id="BradsGadgets-CELLPHONES-2000-V1-R1"
                 subscription-id="SB-BradsGadgets-CellPhones2000-2000-08-23S14-003F9A7C"
                 expiration-date="2001-08-30"
                 quantity="20" >
        <ice-delivery-policy stopdate="2001-08-30" >
          <ice-delivery-rule mode="pull"
                 max-num-updates="720"
                 min-num-updates="13"
                 max-update-interval="P2678400S"
                 min-update-interval="P43200S" />
          <!-- max-num-updates is two per day,
               min-num-updates is 13 per year,
               max-update-interval is 31 days,
               min-update-interval is 12 hours -->
        </ice-delivery-policy >
      </ice-offer>
   </ice-catalog>
</ice-payload>
```

The key thing to observe is that the Syndicator filled in the subscription-id. This is your confirmation that you have a subscription to "Digital Cameras 2000 Newsletter". The other attribute values have been extensively explained in the Syndicator Implementations guidelines above. Basically, it says:

- The Subscriber (you!) will send a requests for updates (mode="pull")
- The subscription ends on the 30 day of August in 2001.
- The Syndicator will update the content at most 720 times or about twice a day.
- The Syndicator will update the content at least 13 times or about once per month.
- The Subscriber should check for new content at most once every 12 hours.
- The Subscriber should check for new content at least once every 31 days.

You may want to consider implementing a simple scheduling mechanism to wake up periodically and check to see if it is time to check for a new newsletter.

2. Implement ice-cancel so subscriptions can be terminated.

   This feature gives the Subscriber the ability to terminate a subscription when desired . Suppose that Joe Cool decides that he no longer desires to subscribe to Brads Digital Camera Newsletter. Joe issues the following payload to Brad:

```
<?xml version="1.0"?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T22:10:33-JoeCool-2321"
     timestamp="22:10:33,741" ice.version="1.1" >
   <ice-header>
      <ice-sender sender-id="4af37b30-2c35-11d2-be4a-204c4f4f5020"
```

```
                  name="Joe Cool" role="subscriber" />
      <ice-user-agent>
         IceBlock Systems ICE Processor, V7.0
      </ice-user-agent >
   </ice-header>
   <ice-request request-id="REQ-2000-08-24T23:47:33-JoeCool-1983" >
      <ice-cancel subscription-id="SB-BradsGadgets-DigCam2000-2000-08-23S14-003F9A7C" />
   </ice-request>
</ice-payload>
```

The key items to note are the subscription-id which tells Brad which subscription to cancel. Brad will verify the sender-id and the subscription-id before cancelling the subscription. If both check out, Brad then sends back the following response:

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-08-24T23:40:45-BradsGadgets-3782"
     timestamp="23:40:45,321"
     location="www.BradsGadgets.com"
     ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
                  name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>
        RockSolid Protocols ICE Processor, V17
      </ice-user-agent >
   </ice-header >
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-9876" >
      <ice-code numeric="200" phrase="OK"
                message-id="REQ-2000-08-24T23:47:33-JoeCool-1983" />
   </ice-response>
</ice-payload >
```

Again, note the importance of the message-id matching the request-id. Also, Brad could send back any of a number of status codes shown in Appendix A.4. For example, the status code "405" indicates an unknown sender and "406" indicates the subscription is unknown. It is straightforward for your implementation to process any of several status codes.

3. Implement ice-change-subscription to adjust subscription parameters.

2.4.4 Enhancements

There are a number of enhancements to this recipe to make your minimal ICE compliant implementation more robust. Collection management will enhance the Syndicators ability to deliver and manage the collection at all of the subscribers. Incremental Update enhances efficiency at both the Syndicator and Subscriber, because it lightens the delivery and processing load. Individual asset repair permits rapid quick repair of a single ice-item without having to download an entire ice-package, again enhancing operational efficiency. Finally, enhancing the delivery rules support means that both Syndicators and Subscribers can better manage server and bandwidth resources and thus obtain significant efficiency.

2.4.4.1 Collection Management

If you haven't done so, consider implementing collection management as described in 2.3.4.3 above.

2.4.4.2 Incremental Update

Once you have implemented basic Collection Management, the first efficiency upgrade is to provide incremental

update. To achieve this, you need to implement `ice-item-group` and `ice-item-delete` to accompany `ice-item` and `ice-item-ref`. With these implemented you can alter your package construction algorithm to add new items and delete items that are no longer used and, most importantly, not send items that your subscriber already has.

2.4.4.3 Individual Asset Repair

This feature allows a subscriber to ask the Syndicator to resend a single ice-item. The value to the subscriber is the savings in processing an entire ice-package to obtain one item and it saves the transmission bandwidth. From the Syndicators view, it is (now) simple to build a one-item response.

2.4.4.4 Enhance Delivery Rules Support

Significant efficiency and performance gains can be made by increasing the number of attributes in the ice-delivery-rule that you support. The suggest order and value is outlined in 3.1 below.

3. Additions & Improvements

There are a number of optional enhancement paths:

- Full Delivery Rule Support
- Simple Negotiation Support
- Full Subscription Negotiation.
- Minimal Subscriber Support.
- Auditing (logging)

3.1 Full Delivery Rule Support

There are many options on delivery rules that enhance the operation of both Syndicators and Subscribers. They are useful to Syndicators in both push and pull delivery mode. In push mode, they determine the best time for package delivery and, in pull mode, they determine if the subscriber is honoring the delivery window. The same comments work in reverse for the Subscriber. In push mode, they determine if the Syndicator is honoring the delivery window. In pull mode, they determine the best time to ask for package delivery from the Syndicator. Here is my (ordered by importance) list of attributes for implementation. (By now you should feel completely free to implement in your own order!)

Many of the parameters concern the idea of a delivery window. The delivery window is the span of calendar time when it is permissable for delivery to occur. For each delivery rule, the time attributes are ANDed in the sense that the delivery window is restricted by each of the time parameters.

mode={"push"|"pull"}
: This has already been implemented. It determines whether the Syndicator pushes content to the Subscriber or whether the Subscriber pulls content from the Syndicator.

url
: The address to send the updates. This effectively allows the subscriber to direct packages to a specific place (or service).

startdate
: This is the start date of the delivery window. If this parameter is left out, the delivery window starts at the `startdate` of the enclosing `ice-delivery-policy`. The value is an `ice-date` (i.e. an ISO8601 date) of the form CCYY-MM-DDTHH:NN:SS,F. CC is the 2 digit century (e.g. 20), YY is

the two digit year (e.g. 00), MM is the two digit month (e.g. 01 is January and 12 is December), DD is two digit day from 01 to 31, HH is two digit hour (00..24), NN is two digit minute (00..59), SS is two digit second (00..59) and F is an optional fraction of a second with up to 9 digits of precision. If the fraction of a second is left out, the seperating comma can also be left out. Note the time 00:00:00 means the start of the day (i.e. just slightly past midnight the previous day) and 24:00:00 means the end of the day (i.e. just slightly before midnight of today). Thus, a `startdate` of 2000-08-27:24:00:00 and a `startdate` of 2000-08-28:00:00:00 are (for all measurable purposes) identical, while a `startdate` of 200-08-28:00:00:00 and a `startdate` of 2000-08-28:24:00:00 are 24 hours apart. *Please note that all ice-times are UTC (or informally GMT) times; so that time computations are normalized to UTC time.* This means for example that if you are in New York, you are 5 hours behind UTC; or 2000-08-28:09:00:00 is at 4 AM your time and not 9 AM. Make sure that you translate accordingly when you present these times to your users. The utility of this to you is that you don't have to adjust times for each interaction with a Syndicator or Subscriber based on their timezone. Everyone using ICE runs on UTC time.

stopdate

This is the stop date of the delivery window. If this parameter is left out, the delivery window stops on the `stopdate` of the enclosing `ice-delivery-policy`. Together the `startdate` and the `stopdate` define the maximum amount of time that the delivery window is open (that is the time that packages may be delivered) for this delivery rule. A day is *open* if package deliveries are possible on any part of the day. The value is an `ice-date` as defined above. However, the `stopdate` must always exceed or equal the `startdate`.

starttime

This is the start time of the delivery window on each day that is open. It is an `ice-time` format and is of the form HH:NN:SS,F where these values are as defined above in `startdate`. The `starttime` defaults to 00:00:00 if left unspecified. A `starttime` of 24:00:00 is legal and effectively begins the adjacent day.

There are two special cases that can arise based on the specification of the `startdate` and the `starttime`. If the `startdate` specifies a time later than the `starttime`, the `startdate` time is used as the `starttime` for that first possibly open day. If the `stopdate` specifies a time that occurs before the `starttime` plus `duration` (see below) runs out, the `duration` is reduced so that the span of time that deliveries can be made ends at the time in the `stopdate`.

monthday

This is a list of the days of the month that the delivery window is restricted to. A month day is expressed as a number from 1 to 31 inclusive or the special tokens, "any" and "last". Note that leading zeros are valid and 01 is the same as 1. A valid attribute is, `monthday="1 15 last"` and means that the delivery window is also restricted to the first, fifteenth and last day of the month between the start and stop dates. If this attribute is not specified, the default value is "`any`" meaning that the delivery window is not restricted by monthday. "`last`" means the last day of the month, so it is 28,29,30,or 31 depending on whether it is leap-year and the current month.

weekday

This is a list of the days of the week that the delivery window is restricted to. A week day is expressed as either a number from 1 to 7 or a named day. Monday is day 1, etc. through Sunday is day 7. The special token "`any`" means that every day of the week is available to the delivery window and is the default value. Thus, a valid weekday attribute is, `weekday="1 3 5 7"` which means that delivery is restricted to Monday, Wednesday, Friday and Sunday. It could also have been expressed as, `weekday="Monday Wednesday Friday Sunday"`.

duration

This is the length of time during a day that delivery can be made and occurs on each open day or part of open day in the delivery window beginning at the `starttime` and lasting for the `duration`, even if it extends into an adjacent open day. This is called the *open time*. The `duration` is specified in `ice-duration` format, that is in number of seconds and fractions of a second in the form PN,FS

where P is the literal 'P', N is a non-negative integer number of seconds, F is a fraction of a second and P is the literal 'P'. The fraction of a second must be readable with up to 9 digits. (You don't have to have 9 digits of precision, use what you have, but you must accept up to 9 digits. We reccommend that you round to your implementations precision.) Thus, P86400S represents 24 hours and is the maximum length of time during an open day that deliveries can be made. Note that for the first and last days, the open time may be reduced. If the time in the `startdate` exceeds the `starttime` attribute, the `duration` is reduced by the difference between the time in the `startdate` and the `starttime` upto the `duration` time. So, if the `startdate` time excceds the `starttime` plus the `duration`, the day is closed on the first day. If the time on the `stopdate` is earlier than the `starttime` plus `duration`, the `duration` is reduced so that the open time stops at the `stopdate` time. Also, if `stopdate` time occurs before the `starttime`, the last day is closed. If the `duration` attribute is not specified, it defaults to 24 hours (P86400S) . A `duration` of 0 hours (P0S) is legal and effectively closes each open day!

min-update-interval
: This is an `ice-duration` and is interpreted according to the role of syndicator or subscriber. If a syndicator, in push mode, it is the minimum amount of time between updates during each open time on each open day. The syndicator will allow at least this amount of time between updates. If a subscriber, it is pull mode, the subscriber will allow at least this amount of time between get package requests during each open time on each open day.

max-update-interval
: This is an `ice-duration` and is interpreted according to the role of syndicator or subscriber. If a syndicator, in push mode, no more than this amount of time will elapse between updates during each open time on each open day. The syndicator promises to update at least once during during this amount of time. If a subscriber, in pull mode, no more than this amount of time will elapse between get package requests to the syndicator. The subscriber promises to ask for updates at least once during this amount of time.

min-num-updates
: This is the minimum number of updates that will occur during each open time on an open day. For "pull" mode, the syndicator is asking the subscriber to issue a get package at least this many times. For "push" mode, the syndicator will deliver a payload at least this many times.

max-num-updates
: This is the maximum number of updates that will occur during each open time on an open day. For "pull" mode, the syndicator is asking the subscriber to issue a get package no more than this many times. For "push" mode, the syndicator will deliver a payload no more than this many times.

For example, the delivery rule that has a start date of 15 September 2000 at 8:00 AM and a stop date of 1 January 2001 at 8:00 AM, with a start time of 7:00 AM and a monthday of 15 and a weekday of "1 2 3 4 5" and a duration of 2 hours means that delivery can occur only on the 15th of the month if it is during the work week of Monday through Friday for two hours from 7 AM to 9 AM. Note that the first open date on the 15th of September, the open time is only 1 hour on that Friday from 8 AM to 9 AM. Note also that the translation to UTC may mean the `ice-dates` are different unless you live in the GMT time zone.

While there is a fair amount of detail here, it is a straight forward implementation of delivery window computation. The value of these delivery rules is that they can assist in a wide variety of dynamic content retrieval that is very difficult to keep up with by hand. For example, if you are monitoring fast changing events such as stock changes, sports action, news stories, entertainment stories or even the weather, automating delivery really enhances your use of this rapidly changing content that would be almost impossible to keep up with by hand. The other potential advantage is that you may be able to schedule delivery during off-peak hours to distribute server loading, and thus control your operational costs.

3.2 Simple Negotiable Subscriptions

Ice defines carefully, the ability to negotiate both operational parameters and any other set of business terms that may interest you. This can be a very powerful addition to your syndication service. To get started on a useful initial implementation, consider permitting your subscribers to fill in the delivery URL in the ice-delivery-rule. This is both easy to do and may require almost no checking on your part. Next consider building the ability to negotiate only a couple of numeric parameters. For example permit negotiating the start time and duration attributes of a delivery rule. This allows your subscribers the ability to restrict delivery to late nights or early mornings and thereby shift the delivery load to lightly loaded times on their servers.

3.3 Full Subscription Negotiation

Once simple negotiation is achieved, Consider implementing multiple parameter resolution. Start by enhancing negotiation to handle enumerations as well as the spans in 3.2 above. Test your implementation by permitting negotiation of the monthday and weekday attributes. Once you have this working, implement the constrained convergence algorithm in both the Syndicator and the Subscriber. Finally, observe that parameter negotiation works on any well defined set of parameters and that they are expressable as an `ice-negotiable`. Thus, your implementation can now support application specific parameter negotiation that has nothing to do with ICE's protocol operational parameters.

3.4 Minimal Subscriber Support

ICE supports a minimal subscriber. This is a subscriber that may not have a server on line at all times. ICE is carefully defined so that the Syndicator never issues a direct request to a subscriber; but always waits for the Subscriber to initiate a conversation. This is because the subscriber may not always be online. Thus, the subscriber is free to completely shut down. However, the Syndicator occasionally needs to obtain information from the subscriber. The Syndicator signals this need by setting a flag in the response to a subscriber when it issues a request. This flag is called the "unsolicited pending" flag. Effectively, it asks the Subscriber to issue an request for these "unsolicited" requests that the Syndicator wants to send. The Subscriber is free to defer this request until it is convenient. Of course the Syndicator is also free to refuse any new requests until the subscriber asks for the "unsolicited" requests and signals this intention with an error code when the subscriber issues a new request. See Appendix A.4 for the status code. The subscriber eventually MUST issue a request for these "unsolicited" requests. Through the use of an unsolicited request, the Syndicator can ask the Subscriber to change a subscription (and possibly re-negotiate it), obtain Subscriber usage logs, obtain the Subscribers view of a subscription, issue a notification, update a collection or ask for collection update confirmation.

3.5 Auditing & Logging

Most robust syndications keep significant logs of their interactions. These are useful to assist in resolving operational problems and to assure correct and desired operation of the syndication and subscription operations. ICE provides a standard means for each side of a syndication relationship to ask for and receive log information. This is also of value for auditing purposes. Often a syndicator will provide content under the requirement of auditing the statistics of the viewers of the content. A useful addition to your implementation is the auditing and logging services of ICE.

4. Conclusion.

Congratulations! If you've implemented an ICE Subscriber and Syndicator using these recipes, you have built a fine piece of software. While there is much more to build, Clearly you have a useful tool to support a wide range of syndication relationships.

Appendices

A. Model Error Responses

A.1 Unknown Subscriber/Syndicator Error Response

The following is a model for responding to a request for which the sender is not known. To populate it correctly for your implementation, you need to replace the values of all of the following attributes:

- payload-id - A sender unique payload identifier.
- timestamp - The time stamp should mark the completion of the ice-payload construction.
- location - The URL where the next request to the sender is to be sent.
- sender-id - This is the UUID of the sender.
- name - This should be the senders Web site URL; or otherwise describe the organization sending error response.
- ice-user-agent content - describes your protocol processor and version number. Use this for debugging the code.
- response-id - A sender unique response identifier (privately generated).
- failing-request-id - The request id from the payload whose header contained the unknown sender.
- ice-code content - The text below is indicative of the type of message you may find useful.

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:03:46-BradsGadgets-3996"
             timestamp="02:03:46,32416"
             location="www.bradsgadgets.com/ice"
             ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
        name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>Road Kill, Inc. ICE Processor, V17</ice-user-agent >
   </ice-header>
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-78889" >
      <ice-code numeric="405"
                phrase="Unrecognized Sender"
                message-id="{failing-request-id}"
      >
        Your sender-id, {unknown-sender-id}, is not known.  Please see,
        http://www.bradsgadgets.com/newsletters/subscriptions to setup
        a subscription.  Or, call {subscription-support-phone} for
        technical assistance.
      </ice-code>
   </ice-response>
</ice-payload>
```

A.2 Unknown Subscription Error Response

The following is a model for responding to a request where the subscription is not known.

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_1.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:03:46-BradsGadgets_3996"
             timestamp="02:03:46,32416"
             location="www.bradsgadgets.com/ice"
             ice.version="1.1" >
   <ice-header >
```

```
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
        name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>Road Kill, Inc. ICE Processor, V17</ice-user-agent >
   </ice-header>
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-78889" >
      <ice-code numeric="405"
                phrase="Unrecognized Sender"
                message-id="{failing-request-id}"
      >
        Your subscription-id, {unknown-sender-id}, is not known.  Please see,
        http://www.bradsgadgets.com/newsletters/subscriptions to setup
        a subscription.  Or, call {subscription-support-phone} for
        technical assistance.
      </ice-code>
   </ice-response>
</ice-payload>
```

A.3 Unimplemented Error Response

This model can be used by both the syndicator and subscriber by replacing the highlighted values with those that your implementation generates.

```
<?xml version="1.0" ?>
<!DOCTYPE ice-payload SYSTEM "http://www.icestandard.org/dtds/ICE1_2.dtd" >
<ice-payload payload-id="PL-2000-07-21T02:03:46-BradsGadgets-3841"
             timestamp="02:03:46,32416"
             location="www.bradsgadgets.com/ice/newsletters"
             ice.version="1.1" >
   <ice-header >
      <ice-sender sender-id="4a2180c9-9435-d00f-9317-204d974e3410"
        name="Brads Gadgets, Inc." role="syndicator" />
      <ice-user-agent>Road Kill, Inc. ICE Processor, V17</ice-user-agent >
   </ice-header>
   <ice-response response-id="RSP-2000-07-21T02:03:45-BradsGadgets-1" >
      <ice-code numeric="503"
                phrase="Not implemented"
                message-id="{unimplemented-request-id}"
      >
        Your request is not implemented in the current syndication program.
      </ice-code>
   </ice-response>
</ice-payload>
```

The message-id in the ice-code matches the request-id of the request that was not implemented.

A.4 List of ICE Codes:

The status values defined by ICE are:

2xx: Success

- 200 OK
  The operation completed successfully.
- 201 Confirmed
  The operation is confirmed.
- 202 Package sequence state already current
  A Subscriber requested a package update, but the Subscriber is already in the current package sequence state, i.e., there are no updates at the moment.

3xx: Payload level Status Codes

These indicate something about the `ice-payload` itself, as opposed to the individual requests and responses within the payload. These codes have one very explicit and important semantic: they are used when the payload could not be properly interpreted, meaning that even if there were multiple requests in the payload, there will be only one `ice-code` in the response. For example, if the payload had been corrupted, it might be so corrupted that it isn't even possible to determine how many requests it contains, let alone respond to them individually.

The specific codes are:

- 300 Generic catastrophic payload error
  Generic status code indicating inability to comprehend the received payload. Usually, it is better to send a more specific code if possible.
- 301 Payload incomplete/cannot parse
  The payload sent is severely garbled and cannot be parsed. For example, if a binary file were sent instead of an XML payload, this would be an appropriate response.
- 302 Payload not well formed XML
  The payload sent is recognizable as XML, but is not well formed per the definition of XML. This is available as both a payload level error and as a request level (4xx) error. Whether a given implementation attempts to interpret not well formed XML so as to generate request level (4xx) errors vs. payload level (3xx) errors is a quality of implementation issue.
- 303 Payload validation failure
  *The payload failed validation according to the DTD. This is available as both a payload level error and as a request level (4xx) error. Whether a given implementation attempts to interpret not well formed XML so as to generate request level (4xx) errors vs. payload level (3xx) errors is a quality of implementation issue. Note that Receivers **SHOULD** perform validation on incoming ICE payloads, but are not required to. Senders **MUST** send only valid ICE payloads or they are in error; however, the ability to detect invalid payloads is a quality-of-implementation issue for the Receiver, and Senders **MUST NOT** assume the Receiver will perform an XML validation on their payloads.*
- 320 Incompatible version
  The ICE protocol version used in the request is not supported. NOTE: The ICE protocol versions are transmitted as part of the payload header, implementations may look there to decide what appropriate corrective actions to take.
- 331 Failure fetching external data
  *The receiver could not follow an external reference (URL) given to it by the sender as an external entity reference. Note that in ICE 1.0 only the Subscriber is permitted to reply with this code. A Syndicator **MUST NOT** reply with this code.*
- 390 Payload temporary redirect
  Used with redirection.
- 391 Payload permanent redirect
  Used with redirection.

4xx: Request level Status Codes

These indicate errors caused by an inability to carry out an individual request. Note that in some cases there are similar errors between the 3xx and 4xx class; the difference is whether or not the error is supplied as a single, payload level error code (3xx) or whether it is supplied as a prerequisite code.

- 400 Generic request error
  Generic status code indicating inability to comprehend the request. Usually, it is better to send a more specific code if possible.
- 401 Incomplete/cannot parse
  The request sent is severely garbled and cannot be parsed. Note that in most cases, a payload level error (301) might be more appropriate.
- 402 Not well formed XML

The request sent is recognizable as XML, but is not well formed per the definition of XML. This is available as both a payload level error and as a request level (4xx) error. Whether a given implementation attempts to interpret not well formed XML so as to generate request level (4xx) errors vs. payload level (3xx) errors is a quality of implementation issue.

- 403 Validation failure
  *The request failed validation according to the DTD. This is available as both a payload level error and as a request level (4xx) error. Whether a given implementation attempts to interpret not well formed XML so as to generate request level (4xx) errors vs. payload level (3xx) errors is a quality of implementation issue. Note that Receivers **SHOULD** perform validation on incoming ICE payloads, but are not required to. Senders **MUST** send only valid ICE payloads or they are in error; however, the ability to detect invalid payloads is a quality-of-implementation issue for the Receiver, and Senders **MUST NOT** assume the Receiver will perform an XML validation on their payloads.*
- 404 This error intentionally left blank
- 405 Unrecognized sender
- 406 Unrecognized subscription
- 407 Unrecognized operation
- 408 Unrecognized operation arguments
- 409 Not available under this subscription
  The Requester has referenced something not covered by the subscription referenced in the request.
- 410 Not found
  Generic error for being unable to find something.
- 411 Unrecognized package sequence state
  The package sequence identifier supplied by the Sender is not understood by the Receiver.
- 412 Unauthorized
- 413 Forbidden
- 414 Business term violation
- 420 Constraint failure
  *Compliant implementations **MUST NOT** send this message if the constraint was not specified in the negotiated subscription.*
- 422 Schedule violation, try again later
  The request was made at an incorrect time. For example, trying to get a package update outside of the agreed upon timing window.
- 430 Not confirmed
  Generic error indicating the operation is not confirmed.
- 431 Failure fetching external data
  *The receiver could not follow an external reference (URL) given to it by the sender. Note that in ICE 1.0 only the Subscriber is permitted to reply with this code. A Syndicator **MUST NOT** reply with this code.*

5xx: Implementation errors and operational failures

These indicate errors caused by internal or operational problems, rather than by incorrect requests. Note that, like all other codes except for the 3xx series, these must be sent individually with each response; if the error condition or operational problem prevents the Responder from resolving the original payload down to the request level, use a 3xx code instead.

- 500 Generic internal responder error
  Catch-all for general problems; recovery/retry behavior unspecified.
- 501 Temporary responder problem
  Too busy, update in progress etc. Eventually an identical retry request might succeed.
- 503 Not implemented
  The server does not implement the requested operation.

6xx: Pending State

These codes indicate a state condition where the Subscriber is expected to send something to the Syndicator, or vice versa.

- 601 Unsolicited messages must be processed now
  The Syndicator has unsolicited messages to send to the Subscriber, and the Subscriber has not yet requested them. The Syndicator has decided (based on implementation specific policy) to refuse to process any more requests until the unsolicited messages are collected.
- 602 Excessive confirmations outstanding
  The Syndicator had requested confirmation of package delivery, and now refuses to perform any additional operations until the Subscriber supplies the confirmations (positive or negative).
- 603 No more confirmations to send
  The Subscriber received an ice-send-confirmations message but believes it has sent all of the confirmations already.
- 604 No more unsolicited messages
  The Subscriber sent an `ice-unsolicited-now` but the Syndicator has no unsolicited messages to send.

7xx: Local Use Codes

These codes are reserved for use by the local ICE implementation and **MUST NOT** ever be sent over the wire. The intent is that this range of codes can be used by the local ICE implementation software to communicate transport level error conditions, or other specific local conditions, using the `ice-code` mechanism in a way guaranteed to not collide with any other usage of `ice-code` values.

9xx: Experimental Codes

ICE implementations **MUST NOT** use any codes not listed in this specification, unless those codes are in the 9xx range. The 9xx range allows implementations to experiment with new codes and new facilities without fear of collision with future versions of ICE.

How a given system treats any 9xx code is a quality of implementation issue.

B. Identifier Implementation Suggestions

B.1 Payload Identifiers

The payload identifier is used to uniquely describe the sender's payloads. One of the best ways to obtain a unique identifier is to use the time coupled with your domain name. The ICE timestamp follows the ISO 8601 time format. So if today is the 23 of August, 2000 at 8:10 am then a reasonable model for a payload identifier is:

PL-[date]T[time]-[domain]-[count]

where

- PL stands for Payload.
- [date] is the date the payload is created. For example, 2000-08-23.
- [time] is the time the payload is created. For example, 08:10:43.007.
- [domain] is the domain name of your ICE site. For example, BradsGadgets.
- [count] is an incrementing number starting at one. Each time a payload is constructed, increment this number. This orders all of the payloads, even if you have a very fast computer that can generate

more than one payload in a millisecond.

Putting it altogether, we get a rather long but unique to Brad identifier:

PL-2000-08-23T08:10:43.007-BradsGadgets-49783

This is just one of the many possible models. You may wish to add additional distinguishing fields such as host name if you have multiple hosts running ICE. You can have almost any format that you want. ICE only requires that the Payload identifier be unique. Since only you have the right to use your domain name, it is a pretty good distinguishing mechanism. Of course, the current time is also a pretty good distinguishing identifier and putting them together is reasonably unique.

B.2 Subscription Identifiers

The subscription identifier is used by the Syndicator to uniquely identify a particular subscription. It is used by a subscriber to tell the Syndicator which subscription is being discussed. Generally, the subscriber needs to keep the subscription information with additional Syndicator information (such as at what domain name to make requests). So for the subscriber, the association is distinguishing only within the subscriptions to a single syndicator. Further only the Syndicator can generate this identifier because it must generate one for each new subscription it creates.

This being the case, the Syndicator could use something of the form:

SB-[domain]-[contentName]-[subscriptionNumber]-[key]

where

- SB stands for Subscription
- [domain] is the domain name of your ICE site. For example, BradsGadgets.
- [contentName] is the name of your content offering. For example, CellPhones2000.
- [subscriptionNumber] is a number that uniquely identifies the subscription within the content offering. For example, the date the subscription started with an incrementing count appended would be: 2000-08-23S14
- [key] is some key that only the subscriber would know and that is hard to guess, say a generated UUID that the Syndicator sends to the subscriber to begin a subscription. For example: 003F9A7C

Putting it all together, you get:

SB-BradsGadgets-CellPhones2000-2000-08-23S14-003F9A7C

Again, you can have almost any format that you want. ICE only requires that the Subscription Identifier be unique across all subscriptions from Syndicator to a receiver so that the Syndicator can distinguish subscriptions; while the above generates a unique subscription identifier across all of a Syndic tors subscriptions.

B.3 Request Identifiers

The Request Identifier is used by the sender to uniquely identify its requests. The receiver uses it only to place into a message-id of the ice-code in a response it makes to the request. This is how the sender knows which response is associated with which request. When the sender receives a response, it looks up the message-id in its list of requests and thereby figures out which request the response if for.

This being the case, the requester (be it Syndicator or subscriber) could use a prototype of the form:

REQ-[domain]-[date]T[time]-[count]

where:

- REQ stands for "Request"
- [domain] is the domain name of your ICE site. For example, JoeCool.
- [date] is the ISO 8601 date. For example, 2000-08-24
- [time] is the ISO 8601 time. For example, 08:10:43.003
- [count] is an incrementing number that increments for each request that is generated. For example, 35478.

Putting it all together, you get:

REQ-JoeCool-2000-08-24T08:10:43.003-35478

Again, you can have almost any format that you want. ICE only requires that the Request Identifier be unique across all payloads from a sender to a receiver so that it can distinguish requests. The receiver is required to place the Request Identifier in the "message-id" of the response.

B.4 Response Identifiers

The response identifier is used by the receiver of a request to uniquely identify its response to the request. When the receiver of a request sends back a response, it become a sender (of the response). The Response Identifier has the same uniqueness requirement that the Resquest Identifier has. That being the case, a similar prototype to the Request Identifier can be used:

RSP-[domain]-[date]T[time]-[count]

where:

- RSP stands for "Response"
- [domain] is the domain name of your ICE site. For example, BradsGadgets.
- [date] is the ISO 8601 date. For example, 2000-08-24
- [time] is the ISO 8601 time. For example, 08:10:43.078
- [count] is an incrementing number that increments for each request that is generated. For example, 1379.

Putting it all together, you get:

RSP-BradsGadgets-2000-08-24T08:10:43.078-1379

You may wonder why this is necessary. For system's of any complexity, you will quickly run into a requirement to log protocol interaction to find and eliminate operational problems quickly and efficiently. The Response Identifier allows you to tie a specific response back to a specific sender and quickly determine, therefore, where miscommunication originates.

B.5 Package Sequence Identifiers

The package sequence identifier(PSID) marks each package sent by a Syndicator to a Subscriber for a subscription. The Syndicator generates all PSIDs. The PSID must be unique within a collection. That being the case, the following can be used for a PSID:

PSID-[collectionName]-[count]

where

- PSID stands for "Package Sequence Idenifier"
- [collectionName] is the name of the collection offered in subscriptions. For example, "CellPhones2000"
- [count] is an incrementing number that increments for each package that is generated. For example, 1379.

Putting it all together, you get:

PSID-CellPhones2000-1379

The Subscriber treats the PSID as opaque, in the sense that all it has to do is store it and compare the one in a new package with the current one.

B.6 Package Identifiers

The package identifier is used by the Syndicator to mark each package. It is unique to the Syndicator. A very simple model for this identifier is:

PI-[time]-[count]

where

- PI stands for "Package Identifier"
- [time] is a compressed date and time. For example, 20000827:095821-234

Putting it all together, you get:

PI-20000827:095821-234

C. Assumptions and Conformance

Each of the recipes in the CookBook are based on the ICE 1.1 Specification. They are designed to begin simply and increase in complexity. Each recipe takes advantage of the the features detailed carefully in the ICE specification. But each recipe is also designed to leave you with a working inter-operable syndication system for processors at the recipe level. To do this, we have selected from the specification conforming simplfing assumptions. The ICE specification is designed to handle a very wide variety of syndication applications and so it has been very careful to make the ICE machinery flexible. This appendix is where we layout those simplifying assumptions. If your application seems to need more capability or seems to require facilities at variance to a recipe, check here first to see if we've made a simplifying assumption that you adjust to your needs. The advantage of checking is so that your implementation can maximally conform to the ICE specification as you build you implementation. This enhances your inter-operability with other implementations. This lowers your over-all syndication costs and it increases the content that you have access to; or the number of subscribers that can access your content. Remember, the value of a protocol standard goes up as the square of the number of users.

This appendix is also where we lay out the major features missing in each recipe that make the recipe not fully ICE compliant. You can use these compliance comments to plan additional features for your implementation; or to compare your implementation to others to determine potential inter-operability issues.

C.1 - Recipe 1 Assumptions and Conformance

This recipe provides excellent service for being so simple. As a Syndicator, all you have to do is create a few XML documents. As a subscriber, all you have to do is read them and fetch (or reference) the content. It provides you with an easy to implement and better means to syndicate content than a simple list of URLs. By simple extension, you obtain lifetime and content protection. But, even better, it positions you to be able to add capability as your growth warrants in a manner that leads to a conforming ICE implementation.

C.1.1 Simplifying Assumptions

This recipe uses the ice-package XML document to describe the content that is available for public use. Almost all of the apparatus available in the ICE protocol is missing. This recipe focuses on the very basics of syndication and uses ICE document formats. It is the first useful step in creating an extensible syndication system. It assumes that subscribers can read and process XML documents and can use other means to access and pull content.

C.1.2 Non-conforming Assumptions

This recipe is not conforming for several basic reasons. The ICE protocol ships an XML document called an ice-payload in a request-reponse manner. This recipe only constructs XML documents called ice-packages (which are contained by ice-payloads) since there is no actual request-response protocol implementation. The ice-package requires several attributes that assist in managing the protocol. These aren't implemented here.

So, this is the beginning of getting the spelling right and getting several core advantages that ICE provides (particularly if you implement the simple extensions). The value of this recipe is that you get the simplest form of syndication; and it permits you to rapidly and simply enhance you syndication/subscriptions beyond the simplest model. And, it gets you ready to enhance your implementation to protocol status, the next recipe.

C.2 - Recipe 2 Assumptions and Conformance

The steps outlined in Recipe 2 model a transaction that has the several restrictions that are valid within the ICE specification. Also, a number of simplifications were made that violate the specification. Nevertheless, this receipe permits you to obtain significant function with a simple implementation. Full inter-operability with compliant ICE implementations is sacrificed as well as basic scheduling. Both scheduling of content and inter-operability capabilities improves your investment in syndication. This can be done in the next step. This recipe provides you with the basic request-response protocol foundation that ICE is built upon. You can use this as a working capability and a foundation for improving to the next level on your way to full ICE function.

C.2.1 Simplifying Assumptions

No Scheduled Delivery.

ICE provides delivery policies in subscriptions that define the scheduling and delivery rules. This allows subscriptions to specify automated delivery. In this recipe, delivery scheduling is done out-of-band.

Fixed Subscriptions

ICE provides an optional ability to select "prototype subscriptions" called offers from a list of offers

in a catalog. Further, ICE provides the ability to negotiate all of the operational parameters of a subscription to arrive at the final subscription. This permits an overall optimal delivery policy for the subscription to be established. This same mechanism can be used to negotiate any set of parameters values of interest to the parties. In this recipe, the operational characteristics of the subscription are set and managed out-of-band. The subscription starts with the syndicator address, subscriber ID and subscription ID fixed in advance as well. While the facility for obtaining offers for subscriptions is required by ICE, the ability to negotiate offer parameters is not required.

Full update only.

ICE permits incremental updates that take advantage of a Syndicators knowlege about its subscriber's collection(s). This permits optimizing content delivery by excluding content already at the subscriber. This implementation resends all elements of the collection, even if the subscriber already has some of the elements.

Delivery by Reference

All content is encoded in `ice-item-refs`. In ICE, content may also be sent inline using an `ice-item`. This recipe provides URL pointers to the content and does not deliver content in-line. This provides the advantage of sending small packets and permits the use of other protocols for delivering the actual content. In the extensions to this recipe, how to send content inline was shown.

No auditing.

ICE provides a feature for both parties to obtain usage logs. This feature is not implemented in this recipe.

Pull delivery only.

ICE permits the syndicator to push content to a subscriber without it being solicited should the subscription so define. This recipe only implements pull delivery. This has the advantage that the subscriber determines its optimal time for content retrieval. However, this time may not be optimal for the Syndicator.

Unstructured Content.

ICE is designed to carry and manage both structured and unstructured content. This recipe does not implement the `ice-item-group` element. This means that the protocol does not know about structured content and delivers it as an opaque whole.

No "Package Processed" confirmation request.

ICE provides two kinds of confirmation, transport delivery and package processed delivery confirmation. Transport delivery is signaled by an ice-code indicating success and acknowlegement of delivery to a receiver. ICE also supports package processed delivery confirmation, which means that all of the items in a package including those delivered by reference have been successfully obtained and processed. This recipe does not implement the syndicator request for package processed delivery confirmation.

No unsolicited messages (or pending flag) sent by syndicator.

ICE supports a simple subscriber that may not be available to receive messages from a syndicator (because it is not in operation.). ICE defines an "unsolicited message pending" flag to inform the simple subscriber that the syndicator would like to send it requests. This facility is not implemented in this recipe.

No individual asset repair request.

ICE supports a mechanism for a syndicator to repair an individual item in a subscriber's collection at the subscribers request. This facility is not implemented in this recipe.

No sending of notification.

ICE supports an optional facilty for a party to send operator notifications to the other party(s). This facility is not implemented in this recipe.

C2.2 Non-conforming Assumptions

No confirmation of package processing response.

ICE requires that a subscriber be able to confirm package delivery processing. This is not

implemented in this recipe. While the facility to send a confirmation request is optional, a full ICE subscriber MUST implement this capability.

`ice-item` unsupported.

ICE requires that a subscriber be able to process inline content as well as content by reference. `Ice-item` processing is not implemented in this recipe. However, it is a straight-forward extension as explained above.

Unsolicited messages and unsolicited message pending flag unsupported.

ICE requires that a subscriber be able to process unsolicited messages as well as honor the unsolicited message pending flag. This subscriber does not implement these facilities.

individual asset repair response unsupported.

The Syndicator must implement an ability to respond to requests for item repair. This is not implement is this recipe.

Notification receipt unsupported.

Neither the syndicator nor the subscriber implement notification receipt procesing in this recipe as is required by ICE. While the facility to send a notification request is optional, all ICE processors MUST implement notification receipt processing.

NOP unsupported.

ICE requires that a Syndicator and subscriber be able to receive and process a NOP. This implementation does not support NOP. This is a nearly trivial extension to this recipe.

Catalog features unsupported.

This recipe does not implement support for the`ice-get-catalog` request and an `ice-catalog` response. Every ICE Syndicator MUST provide an `ice-catalog` of `ice-offers` which are prototype subscriptions in response to an `ice-get-catalog` request. Every subscriber MUST be able to request an ice-catalog with an ice-get-catalog request and choose an offer. Further, the subscriber MUST be able to send an ice-offer to a Syndicator and the Syndicator MUST be able to send an ice-subscription to a subscriber. This interchange is NOT implemented in this recipe.

C.3 - Recipe 3 Assumptions and Conformance

This Recipe enchances the basic request/response protocol by adding additional operational facilities that are part of the ICE specification.

C.3.1 Simplifying Assumptions

No Scheduled Delivery.

ICE provides delivery policies in subscriptions that define the scheduling and delivery rules. This allows subscriptions to specify automated delivery. In this recipe, delivery scheduling is done out-of-band.

Fixed Subscriptions

ICE provides an optional ability to select "prototype subscriptions" called offers from a list of offers in a catalog. Further, ICE provides the ability to negotiate all of the operational parameters of a subscription to arrive at the final subscription. This permits an overall optimal delivery policy for the subscription to be established. This same mechanism can be used to negotiate any set of parameters values of interest to the parties. In this recipe, the operational characteristics of the subscription are set and managed out-of-band. The subscription starts with the syndicator address, subscriber ID and subscription ID fixed in advance as well. While the facility for obtaining offers for subscriptions is required by ICE, the ability to negotiate offer parameters is not required.

Full update only.

ICE permits incremental updates that take advantage of a Syndicators knowlege about its subscriber's collection(s). This permits optimizing content delivery by excluding content already at the subscriber. This implementation resends all elements of the collection, even if the subscriber already has some of the elements.

No auditing.
ICE provides a feature for both parties to obtain usage logs. This feature is not implemented in this recipe.

Unstructured Content.
ICE is designed to carry and manage both structured and unstructured content. This recipe does not implement the `ice-item-group` element. This means that the protocol does not know about structured content and delivers it as an opaque whole.

No unsolicited messages (or pending flag) sent by syndicator.
ICE supports a simple subscriber that may not be available to receive messages from a syndicator (because it is not in operation.). ICE defines an "unsolicited message pending" flag to inform the simple subscriber that the syndicator would like to send it requests. This facility is not implemented in this recipe.

No individual asset repair request.
ICE supports a mechanism for a syndicator to repair an individual item in a subscriber's collection at the subscribers request. This facility is not implemented in this recipe.

Trivial Negotiation Only
ICE supports the apparatus to negotiate the parameters of a subscription. Both operational parameters and other business terms defined in an offer can be negotiated. This recipe selects the simpler "trivial" negotiation mechanism, the minimum required by the Specification.

C3.2 Non-conforming Assumptions

Unsolicited messages and unsolicited message pending flag unsupported.
ICE requires that a subscriber be able to process unsolicited messages as well as honor the unsolicited message pending flag. This subscriber does not implement these facilities.

individual asset repair response unsupported.
The Syndicator must implement an ability to respond to requests for item repair. This is not implement is this recipe.

Catalog features unsupported.
This recipe does not implement support for the`ice-get-catalog` request and an `ice-catalog` response. Every ICE Syndicator MUST provide an `ice-catalog` of `ice-offers` which are prototype subscriptions in response to an `ice-get-catalog` request. Every subscriber MUST be able to request an ice-catalog with an ice-get-catalog request and choose an offer. Further, the subscriber MUST be able to send an ice-offer to a Syndicator and the Syndicator MUST be able to send an ice-subscription to a subscriber. This interchange is NOT implemented in this recipe.

C.4 - Recipe 4 Assumptions and Conformance

A number of simplifying assumptions have been made in the CookBook to get a conforming but not complete roadmap for implementing ICE.

C.4.1 Simplifying Assumptions

No Scheduled Delivery.
ICE provides delivery policies in subscriptions that define the scheduling and delivery rules. This allows subscriptions to specify automated delivery. In this recipe, delivery scheduling is done out-of-band.

Fixed Subscriptions
ICE provides an optional ability to select "prototype subscriptions" called offers from a list of offers in a catalog. Further, ICE provides the ability to negotiate all of the operational parameters of a subscription to arrive at the final subscription. This permits an overall optimal delivery policy for

the subscription to be established. This same mechanism can be used to negotiate any set of parameters values of interest to the parties. In this recipe, the operational characteristics of the subscription are set and managed out-of-band. The subscription starts with the syndicator address, subscriber ID and subscription ID fixed in advance as well. While the facility for obtaining offers for subscriptions is required by ICE, the ability to negotiate offer parameters is not required.

Full update only.

ICE permits incremental updates that take advantage of a Syndicators knowlege about its subscriber's collection(s). This permits optimizing content delivery by excluding content already at the subscriber. This implementation resends all elements of the collection, even if the subscriber already has some of the elements.

No auditing.

ICE provides a feature for both parties to obtain usage logs. This feature is not implemented in this recipe.

Unstructured Content.

ICE is designed to carry and manage both structured and unstructured content. This recipe does not implement the `ice-item-group` element. This means that the protocol does not know about structured content and delivers it as an opaque whole.

No unsolicited messages (or pending flag) sent by syndicator.

ICE supports a simple subscriber that may not be available to receive messages from a syndicator (because it is not in operation.). ICE defines an "unsolicited message pending" flag to inform the simple subscriber that the syndicator would like to send it requests. This facility is not implemented in this recipe.

No individual asset repair request.

ICE supports a mechanism for a syndicator to repair an individual item in a subscriber's collection at the subscribers request. This facility is not implemented in this recipe.

Trivial Negotiation Only

ICE supports the apparatus to negotiate the parameters of a subscription. Both operational parameters and other business terms defined in an offer can be negotiated. This recipe selects the simpler "trivial" negotiation mechanism, the minimum required by the Specification.

C4.2 Non-conforming Assumptions

individual asset repair response unsupported.

The Syndicator must implement an ability to respond to requests for item repair. This is not implemented is this recipe.