Content Delivery Subscription System

Andrew Weinrich
10 June 2003

This document describes a generic protocol for managing subscriptions of discrete content published by hosts. It is built upon the Secure Multipoint Message Delivery Protocol, and provides the automated key exchange system that SMMDP lacks. It also automates the subscription process, and is optimized for simplicity by storing a minimum of data at the hosts.

Overview

CDSS is designed for a network of hosts that both produce information and consume it in the form of discrete "events". This does not necessarily imply the meaning of events in the GUI/object-oriented sense, but rather content as single, asynchronous, independent blocks, as opposed to streamed or interactive data.

Every host in the network is equal (real-world differences in size and importance notwithstanding), being able to both generate events and receive them from other hosts. At first, this would suggest a pure "peer-to-peer" model, in the Gnutella mold.

However, it is likely that some of these events will contain sensitive information that publishers will want to send only to those who should receive it. Also, the potential size of the network is very large, and every host should be able to receive events from every other host; the scaling problems of P2P networks that attempt global awareness are well-documented.

Instead, CDSS hybridizes centralized publishing and P2P in a manner similar to the Napster network: a central registry server that maintains an index of "metadata" on all hosts in the system, and the individual hosts themselves, which communicate in a "peer-to-peer" system. The actual data exchange is between the hosts themselves, while the server only acts as a directory to help the hosts find each other.

Security

CDSS uses the Secure Multipoint Message Delivery Protocol for communication. SMMDP provides the security necessary to transmit sensitive event data, while CDSS implements an automated key exchange system to make SMMDP more usable.

System Roles

For purposes of this discussion, we will assume three participants:

- **Server (M)**

This server is responsible for maintaining all the records of subscriptions, as well as all the public keys used for signing and delivering messages.
The subscription server must maintain a published public key used for communication with hosts in the network. Hosts may send their public keys to the server in the message body of the first communication establishing an account.

- **Publisher (P)**
  A host that generates events to send out to subscribers.
- **Subscriber (S)**
  A host that wishes to receive the Publisher's Events.

It is important to note that the Publisher and Subscriber roles can, and should be, interchanged. All hosts in the network (save the Server) can potentially both publish and subscribe to events.

All hosts, the server included, are identified in communications solely by their IP addresses.

Use of SMMDP

The CDN Subscription system uses the Secure Multipoint Message Delivery Protocol for all communication:

- Requests from hosts to the server (such as subscription activations, revocations, and subscriber lists queries) are synchronous, replied, and reliable.
- Transmission of events between hosts may be reliable or unreliable depending on the administrator's configuration. Events will usually be sent to multiple hosts, though the actual method (separate TCP communications, IP-level multicast, etc) will depend on the network environment.

Communication System

Host-Server Commands

The following are requests that a host may send to the server. Authentication is performed automatically by the security features in SMMDP, so no login/password is necessary. Several terms are used to describe the psuedo-code API:

- **eventType**
  Refers to an enumerated class of events. This value is used as the SMMDP "function code".
- **publisher**
  The identifier (IP address) of a host in the publisher role.
- **subscriber**
  The IP address of a host in the subscriber role.
- **transmissionKey**
  A public key that should be used to encrypt the SMMDP session key when sending events to a subscriber.
- **signingKey**
  The key that a publisher will use to authenticate SMMDP message hashes.

- **filter**
  Some set of parameters that controls the results returned by the request.

For most of these commands, the server will respond with a result message that merely indicates success of failure. Other responses are noted below.

The following three commands manage subscriber actions:

- **Subscribe(publisher, eventType, transmissionKey)**
  Request a subscription to a type of event from a publisher. Includes the key which is to be used for sending the data to the host. If the subscription is allowed, the server will respond with a message that includes the publisher's signing key
- **Unsubscribe(publisher, eventType)**
  Reverses the subscription established by Subscribe. The server response will indicate success or failure (for example, if the host was not subscribed to begin with).
- **GetSubscriptions(filter)**
  Returns a set of three-tuples (publisher, eventType, signingKey) that describe all the publications to which this host is currently subscribed, including the keys that the publishers will be using to sign the transmissions.
  The host may filter the request based on certain publishers or classes of events.

These requests are used for publisher management:

- **AllowSubscriber(subscriber, eventType, signingKey)**
  Allows a host to subscribe to a certain type of event, and specifies the key that the publisher will be using to sign the message.
  Note that this request does not actually activate the subscription; the subscriber must subsequently make a request to receive the specified type of events from this publisher.
- **RevokeSubscriber(subscriber, eventType)**
  Reverses the action of AllowSubscriber; prevents a host from receiving the specified type of events.
- **GetSubscribers(filter)**
  Returns a set of three-tuples (subscriber, eventType, transmissionKey) that describe which hosts have subscriber to which events from this publisher, and the transmission keys that the publisher should use to communicate with them. The publisher can filter the request on certain event types or hosts).

Example communication

The following chain of events would establish a subscription between S and P for event class E:

1. Offline, the administrator of S contacts the administrator of P and asks for a subscription to events of class E, which P grants.
2. P sends a request to M to allow S to subscribe to E events, and states that it will be signing with the key KpubP
3. M records this allowance, and replies affirmatively to P.

4. Now that the subscription is allowed, S sends a request to M to subscribe to P's E events, and gives it a public key of predetermined length, KpubS.
5. M checks to see whether the subscription is allowed. If not, it replies negatively. If so, it activates the subscriptions and stores KpubS as associated with that subscription, then sends an affirmative reply including P's signing key, KpubP.
6. After a certain periodic interval, P contacts M, and requests a list of all the subscribers to certain classes of events, including class E.
7. M returns a list of all the subscribers, and their keys, for the requested event classes. In this case, it returns the (event, subscriber, key) three-tuple of (E, S, KpubS).
8. Using the secure delivery protocol, P constructs a message that is destined to at least S, with the session key encrypted with KpubS, the message data and hash encrypted with the session key, and the message hash decryptable with KpubP; the message is then sent at a predetermined level of reliability.
9. When S receives the message, the secure delivery protocol finds its copy of the session key in the message, and decrypts it using KprivS. The session key is then used to decrypt the body of the message, including the signed hash. The hash is decrypted
10. The protocol library returns the event data to S.

Subscriptions can be revoked or deactivated, or keys changed at the server, using similar requests.

Drawbacks

Trust issues

All the trust lies in the server. If the server is compromised, it may send false lists of subscribers to publishers, ignore requests, and generally behave erratically. However, because no actual event data is sent through the server, it cannot intercept private data or modify it for subscribers. Additionally, due to the use of exclusively public keys at the server, there is no possibility that a compromised server could use a stolen key to eavesdrop.

The initial communication between S and P to establish that a subscription is allowed is not handled by this system. It is assumed that the communication will take place between the system administrators in some "real-world" way: phone, email, fax, etc.

Efficiency

It is possible that some hosts will see it as comfortable or desirable to only maintain one public key for communication with all other hosts, or for all classes of events with one host. In this case, that same key will be replicated in the subscription server for every event/host combination, a geometric increase in storage requirements.

Depending on the frequency of publishing events, and the number of recipients, there may be a fair duplication of effort in continually sending copies of the same keys to all

the publishers from the server, when these keys and lists of recipients may not change very often.

These two problems are simply the result of a tradeoff against the inherent problems of cache consistency across a wide network. The model described here aims for maximum simplicity, storing everything in the subscription server and remembering nothing at the hosts. A real-world implementation may find it advantageous to cache keys and other data, and only propagate update information.