Description of the ITX Signaling Package

Donna Bergmark

August 12, 1999

1 Introduction

ITX signaling is handled by the `cnrg.itx.signal` package. Each ITX application has exactly one *DesktopSignaling* object, which handles all signaling on behalf of the application. We would like to make ITX signaling SIP [2] compliant. The first step is to understand how the package currently works.

2 Functions supported by Signaling

The `cnrg.itx.signal` package takes care of generic telephony control plane functionality; the gateway and pbx packages take care of specialized signaling functionality. The following basic services are provided by the `cnrg.itx.signal` package, and are described in more detail in the remainder of this report:

- DesktopSignaling **constructor** - provides a signaling object for your ITX application. This object talks to the corresponding object in a peer ITX application. In this paper, the term *DesktopSignaling* denotes the signaling object belonging to a given ITX application.
- **Call setup and teardown** - one ITX application calls another one, or calls a real telephone. Call setup negotiates the establishment of a call. When the call aborts, or a participant hangs up, signaling tears down a call, releasing resources.
- **Keepalive** - signaling objects send out periodic keepalive signals to their peer signaling objects, and detects cases when the peer has died.
- **Registration**(optional) - the signaling object registers the ITX application upon launch, and handles authentication of the application. Upon termination, *DesktopSignaling* deregisters the application from the ITX database.

- **Data Connection Setup** - the signaling object takes care of setting up a complete audio connection with the peer application, but does not open it. However, upon termination, *DesktopSignaling* does close the connection.

An important aspect of the signaling component is that it uses an event model to handle the complex logic of changes in call state. Section 4 describes this event model.

3 The *DesktopSignaling* Object

Application Specific Information		
myUID	UserID	Email address or application name under which the application is registering itself
myPassword	Password	Password being used by the application to authenticate itself
myDesc	String	Optional description of the application using this *DesktopSignaling*
myName	String	String representation of the IP address at which this *DesktopSignaling* is waiting for input from other *DesktopSignaling* objects
Object Handles		
myDirS	DirectoryService	The *DirectoryService* object being used by this application and this *DesktopSignaling*.
myApp	SignalingObserver	The application on whose behalf we are handling the signaling function. This is the SignalingObserver that will handle any signaling events that arise. Typically `myApp` is the object that instantiated this *DesktopSignaling*.
myDSS	DesktopSignalingServer	This is the runnable object that listens for incoming packets and disposes of them appropriately
Internal Variables		
myConnectionList	Hashtable	Keeps track of peer applications to whom we are connected.
myConnSeqNumber	long	A sequence number to use when we are creating a new *SignalConnection*.

Location Descriptors		
myIAdd	InetAddress	The IP address at which this *DesktopSignaling* is listening for incoming packets.
myLoc	Location	This is our IP address and port number, in Directory Service format. In general, a Location can also be an email address or telephone number, but for *DesktopSignaling* objects, it is always an IP address.
myServerPort	int	The port on which this *DesktopSignaling* is listening for incoming packets.

Miscellaneous		
TIMEOUT	int	The constant declared in `cnrg.itx.signal.DesktopSignal` as the number of milliseconds to wait for responses to signaling packets. Currently this is set to 20 seconds. After 20 seconds with no response, the peer application is declared dead and the connection is removed from `myConnectionList`. It is also used by `myDSS` to set a limit on how long to block while waiting for an incoming connection request. The TIMEOUT can be changed via the `setTimeout()` method.

Some of these fields are described below in further detail.

3.1 Directory Service

Each ITX application has exactly one *DesktopSignaling* object, and each *DesktopSignaling* object has at most one *DirectoryService* object, obtained at *DesktopSignaling* instantiation time. If the ITX application is being run without a DirectoryService, then all of its calls must be to explicit ports at explicit IP addresses. If the Directory Service is being used, then one may call telephones, other users (at their email address), or other applications (by name).

For a *DesktopSignaling* to get a *DirectoryService* for the application, *DesktopSignaling* needs a UserID, Password, and optionally a configuration file, all of which are passed to the *DirectoryService* constructor. `myDirS` is used by DesktopSignaling to look up other users:

```
LocationList ll = myDirS.getLocationListByID(u);
```

where `u` is the UserID of somebody our application is trying to call. `myDirS` is valid (usable) or not depending on whether `myUID` and `myPassword` passed the authentication test.

If the application needs to do some directory service stuff directly, it can get a handle to `myDirS` by calling `DesktopSignaling.getDirectory()`. The *DesktopSignaling* object not only takes care of authentication, but also registers the current roaming location of the user in the database:

```
myDirS.registerLocation(Location.ROAMING, myLoc);
```

3.2 Signaling Observers

As explained further in Section 4, it is up to the application to provide event handlers if it wants to do something special for invitations and other events caused by peer applications. SignalingObserver is the interface that must be implemented, and AbstractSignalingObserver is the default implementation, in which all invitations are handled by doing nothing. `myApp` is the handle by which *DesktopSignaling* invokes the application's signal event handlers.

In addition, if ITX applications want to make non-blocking calls (for example, to be able to attempt calling a number of locations in parallel) then the application must also implement the SignalConnectionObserver interface, to track the progress of the call.

3.3 Desktop Signaling Server

Exactly one *DesktopSignalingServer*, `myDSS`, is created for each *DesktopSignaling* object. `myDSS` is a thread which listens on a server socket for incoming packets (it sets up its own server socket, letting Java choose the port number). This port number is saved in `myServerPort`. As can be seen from the following, `myDSS` is what causes `myApp`'s signal event handlers to be called.

The thread enters an endless loop, accepting incoming connection requests. Whenever a request arrives, `myDSS` spawns a SigPacketHandler thread with the server socket on which the request was made, an incremented request number, and a handle to *DesktopSignaling*. The endless loop continues until `killServer()` is called during the `logout()` process.

The *SigPacketHandler* reads in and decodes a signal packet, treating the socket as an ObjectInputStream.[1] In `SigPacketHandler.run()` we have:

```
mySP = (SigPacket) myIn.readObject();
```

where `myIn` is the input stream of the socket.

[1]All packets exchanged by ITX peers are *SigPacket*s, which are serialized to and from object streams. *InvitePacket* is a subclass of *SigPacket*.

Upon decoding the packet (which *must* be an *InvitePacket*[2]), the *SigPacketHandler* constructs a *SignalEvent* and calls the appropriate method in *DesktopSignaling*:

Kinds of Packets and Methods Called		
Method ID	method called	event
DIAL	`handleDialInvite`	*InviteSignalEvent*
HANGUP	`handleHangupInvite()`	*HangupSignalEvent*
SENDDTMF	`handleSendDTMFInvite()`	*DTMFSignalEvent*
ALIVEQUERY	`handleAliveInvite()`	*AliveSignalEvent*

At this stage, *SignalEvent*s contain only two things: the *InvitePacket* and a handle to *DesktopSignaling*.

The SigPacketHandler thread then dies, while further processing of the *InvitePacket* proceeds in *DesktopSignaling* and then in `myApp`. In the meantime `myDSS` could well field another request. We rely on the Java scheduler to run the handlers concurrently.

3.4 Connection List

The objects contained in `myConnectionList` are *SignalConnection*s. The connection list is what allows an ITX application's single *DesktopSignaling* object to maintain many connections to the outside world at the same time.

The only time a connection gets added to the table is when our application invokes the `Dial()` method and our INVITE is accepted by the peer, or when we receive an incoming INVITE packet and the peer confirms our ACCEPT. The only time a *SignalConnection* is deleted from the table is when our peer hangs up on us, or our application invokes our `Hangup()` method with some signal connection. Each *SignalConnection* contains two sequence numbers: ours and our peer's. We use our sequence number as the hash key for `myConnectionList`. (Hashtable `get()` and `remove()` methods are sufficient to maintain `myConnectionList`.)[3]

3.5 Signal Connection

The *SignalConnection* object results from a successful `Dial()` or comes to us in an INVITE packet. It contains sequence numbers, an audio connection, stuff about the peer, a KeepAlive thread, a Dialer (if relevant), and the current state of the call. Initially, a *DesktopSignaling* object has no signal connections at all. The standard *SignalConnection* has the first three fields

[2]that is, `mySP.isInvitePacket()` is true

[3]As of July 27, 1999 these methods are synchronized to remove a race condition among multiple non-blocking dial operations.

of the following table filled in, and the asynchronous dialed connection has all five fields filled in. The default *SignalConnection* has no data connection and no peer.

Field	Type	Description
myC	Connection	The audio connection used in this *SignalConnection*
myPeerIP	String	IP address of our peer *DesktopSignaling*
myPeerPort	int	Port number on which our peer is listening
myPeerUID	UserID	Use this to look up the peer application in Directory Service
myDialer	DialThread	The thread that dialed or is dialing or will dial this *SignalConnection*

The primarily methods on a *SignalConnection* are field accessors and setters, `open()` and `close()`, `startCall()` to start a call to a peer and `abortCall()`, and `startKeepAlive()`.[4] The *SignalConnection* is used primary as a bundle of data, containing all the state of the call.

4 Event Model

ITX uses the events and listeners model. Upon instantiating a *DesktopSignaling* object, the application must specify a listener for signal events, typically itself. The application can also be a listener for signal connection events.

An ITX application can implement `SignalingObserver`, which is an interface, or it can extend `AbstractSignalingObserver`, which is an adapter, or it can pass `AbstractSignalingObserver` as a handle when instantiating a *DesktopSignaling*. Typically, an ITX application passes itself:

```
mySig = new DesktopSignaling(this, ...
```

When changes occur in the environment, an event is generated; see the subpackage `cnrg.itx.signal.SignalE` for details. After constructing the event, *DesktopSignaling* calls one of the application handlers. There are seven signal events in all. Here are the events and their handlers:

[4] Note that hangups are a function of the *DesktopSignaling*, and not of the *SignalConnection*.

Signal event	Handler	Description
AbortSignalEvent	onAbortCall()	tells an app to abandon call setup
AliveSignalEvent	handleAliveEvent()	(internal to *DesktopSignaling*)
DialSignalEvent	onAccept()	tells app something happened
	onReject()	during non-blocking Dial()
	onBusy()	
	onInvalid()	
	onTimeout()	
	onIncompatible()	
	onError()	
DTMFSignalEvent	onDTMF()	tells app that DTMF has been received
HangupSignalEvent	onHangup()	tells app that peer has hung up
InviteSignalEvent	onInvite()	tells app that peer is trying to call it
SignalEvent	(none)	base class for other signal events

By time the application handler is called, a third item - the signal connection – has been added to the *SignalEvent*. Thus the *SignalEvent* now contains three significant objects: an *InvitePacket*, a *SignalConnection*, and the *DesktopSignaling* that generated the event.

The application specifies its disposition of invitation events by calling a method on the *SignalEvent*. For example:

```
ise.accept();
```

There is an *InvitePacket* inside that *SignalEvent*, and its `resultFlag` is set to `SignalID.ACCEPT`, flagging the packet as accepted.

To recap, *InvitePacket* extends *SigPacket* and is serializable. It is part of a *SignalEvent* object, which is contained in a *SignalConnection*, along with a data connection (*Connection*).

5 Call Setup and Teardown

Setting up a call begins when one ITX application decides that it wants to make a call to another party. The ITX application begins the process by invoking the `Dial()` method on its *DesktopSignaling*. The parameters to the `Dial()` method determine who the callee is and what the characteristics of the resulting audio connection should be.

Like SIP, ITX uses a three-way handshake between peers in order to set up a call. The successful connection is established in the following three steps:

1. Caller[5] sends an INVITE packet to callee, containing caller's location, and caller's side of the data connection.

2. Callee sends back an ACCEPT packet containing a negotiated data connection having the callee's desired side. At that point, the callee has a complete data connection object, with both its and callers connection properties attached to it.

3. Caller gets the ACCEPT packet, determines that the negotiated connection is feasible, and returns a CONFIRM packet to the callee. Caller now has a complete data connection object, and returns it along with other information to the `onStart()` method of its SignalingObserver (i.e. the application that originally invoked the `Dial()`). It also adds the *SignalConnection* to `myConnectionList`.

The 3-way handshake can be aborted at the following junctures:

- Callee sends a REJECT packet, meaning it does not want to accept caller's invitation, period.[6] Upon receiving the REJECT, the `Dial()` method throws an InviteRejectException which can be caught by the application.

- Callee sends a BUSY packet meaning that the Location is busy. *DesktopSignaling* tries the next Location in the LocationList (if blocking Dial) else constructs a DialSignalEvent and calls an application handler, `onBusy()`.

- No packet at all came back from the Callee, even though its *DesktopSignaling* appears to be running. This can happen when a user is using a Desktop Virtual Telephone but has gone to lunch and does not see the "ring". If this was the last Location in the list, the connection is torn down and `Dial` throws an InviteTimeoutException, otherwise the next Location is called.

5.1 Directory Lookup

ITX applications invoke `DesktopSignaling.Dial()` to reach somebody. If no Directory Service is available, then the location (IP address and port number) must be given as a parameter. Otherwise, the parameter can be a string representing who is being called. In the current version of ITX, this string can be one of the following:

[5]The *caller* is the *DesktopSignaling* on which the `Dial()` method was first invoked. The *callee* is the *DesktopSignaling* belonging the the Gateway or other ITX application, depending on which *Location* of the target is being contacted.

[6]For example, if the caller is on a computer which has no audio devices, then the caller's *DesktopSignaling* will reject the invite and then call its application's `onAbortCall` handler.

1. an email address
2. a telephone number
3. userID (typically the name of some server application)

DesktopSignaling uses this to get a *LocationList* from `myDirS`. The *LocationList* begins with the user's roaming location (if any), followed by a list of other dynamic locations, ending with a list of quasi-static locations at which the user can be reached.

DesktopSignaling tries each location in turn until the INVITE gets back an ACCEPT or until the end of the list is reached. If the location is another user in the database, *DesktopSignaling* will recurse to one level.

5.2 Setting up the Data Connection

If the targeted callee answers at some location and agrees on a suitable data connection between us and it, then the connection is successful. `Dial()` returns a *SignalConnection* to represent the call that is in progress. A *SignalConnection* is basically a data connection plus some signal data (see Section 3.5). The default data connection is `NetworkSource --> SpeakerDestination` for input and `MicrophoneSource --> NetworkDestination` for output. The audio connection is set up (i.e. resources allocated) but not opened. It is up to the calling application to actually open the signal connection.

The application can override the input and output channels by passing them as parameters to the `Dial()` method. In particular, half-duplex applications might choose to specify `(Channel)null` as one of the channels. Either both channels must be given in the `Dial()` call, or neither.

5.3 Call Teardown

A call, represented by a *SignalConnection*, can be terminated in one of 5 ways:

1. *DesktopSignaling*'s peer sends it a HANGUP request for a given *SignalConnection*.
2. The application invokes `Hangup()` on *DesktopSignaling*, passing it the *SignalConnection* involved.
3. A *SignalConnection's* KeepAlive thread fails to open a socket to the peer application (see Section 6).
4. The application invokes `isAlive()` on *DesktopSignaling*, passing it a *SignalConnection*, and the attempt to send an "alive" INVITE on this connection fails in some way.

5. The application invokes `abortCall()` on *DesktopSignaling*, passing it the *SignalConnection* to be aborted.

In each case, *DesktopSignaling* closes the *SignalConnection*, which releases all the audio resources associated with the call. In most cases the *SignalConnection* is also removed from `myConnectionList`.[7]

6 Keepalive

In ITX, signaling is bound into the application. There is no central signaling daemon that keeps track of which processes are alive and which are not. This "shared-fate" scenario makes it simple for the control layer to keep track of its peer processes by using a "hello" protocol.

A *SignalConnection* has a KeepAlive thread once its `startKeepAlive()` method is invoked. A *DesktopSignaling* calls this method during the `Dial()` sequence if its INVITE has been accepted by its peer[8]:

```
sc = (SignalConnection) myConnectionList.get(seq);
sc.startKeepAlive(this);
```

or when our ACCEPT of an INVITE has been CONFIRMed:

```
sc = new SignalConnection(...);
:
myApp.onStartCall(sc);   // does app want to start the call?
sc.startKeepAlive(this); // if onStartCall returns
```

The call to `startKeepAlive()` causes a *KeepAlive* object to be created, which is a thread that periodically opens a socket to a peer. If the socket open fails three times in a row, the peer is assumed to be dead; *DesktopSignaling*'s `handlePeerNotAlive` method is invoked to shut down the connection.

The thread is stopped when our application calls our `Hangup()` method on some active signal connection, or when *DesktopSignaling* receives a HANGUP packet from its peer:

```
KeepAlive ka = sc.getKeepAlive(); // sc is SignalConnection
if (ka != null) ka.cleanup();
```

[7] The exception is `abortCall()` which simply marks the *SignalConnection* IDLE.

[8] For non-blocking `Dial()`s, it is a DialThread that calls the method but that is only a detail.

BUG: I believe that the call to `ka.cleanup()` should be moved into `SignalConnection.close()`, since the KeepAlive thread is part of the *SignalConnection* object.

BUG: Currently we have a bug in signal connection's implementation of `abortCall()` in that the signal connection stops its DialThread but not its KeepAlive. I think that if a signal connection is in the IDLE state, it should also have been closed or never have been opened.

The other objects related to KeepAlive are *AliveSignalEvent* and its handler *handleAliveInvite()*. If an application wants to check to see whether its peer is still alive, it can invoke the `isAlive(SignalConnection)` on its *DesktopSignaling*. This will cause an "alive" INVITE packet to be created and sent to the peer.

If successful, the peer *DesktopSignaling* object will invoke its `handleAliveInvite()` method, which returns a CONFIRM packet. Otherwise, *DesktopSignaling* has inadvertently discovered that the peer is dead and will treat this via its `handleHangupInvite`, which closes the signal connection. That is, discovery of peer death is treated as a HANGUP request.

7 Registration

The ITX directory keeps track of all applications that are currently running, as well as users of ITX. Users and applications have unique userids, passwords, and level of authorization.

When a *DesktopSignaling* object is first created by an ITX application, some of its parameters are the userid and password under which the application is authenticating itself. For example, when CUPS begins running, it provides its userid (*adm*) and password to the DesktopSignaling constructor, which in turn obtains a *DirectoryService* and calls `DeclareIdentity()` on it. If successful, further directory services are allowed.

As a side effect, this *DesktopSignaling*'s current IP address and port number (on which it is listening) are registered in the database. This will allow other ITX applications to send Dial invitations to this *DesktopSignaling*. This location is known as the user or application's "roaming location".

If a user logs into CUPS, since CUPS is running with management-level privileges, CUPS can obtain the user's NETid and ITX password and use them to register the user's current roaming location, which again, is the IP address and port number on which CUPS' *DesktopSignaling* is listening for invites.

8 Gateway and PBX

The Gateway and PBX are part of signaling, even though they are in their own packages (`cnrg.itx.gtwy` and `cnrg.itx.gtwy.pbx`). The Gateway manager is a special ITX application which gets a *DesktopSignaling* just like any other ITX application.

The difference is that the Gateway extends the signaling interface to handle real telephone numbers and the PSTN, with or without the PBX package.

If a CUPS user types in a real telephone number rather than the name of another ITX user or application, then CUPS will send a call invitation to the gateway (it finds out where the gateway is running by looking up ITX application *gatewaysrv* in the database). The Gateway's *DesktopSignaling* is listening for invitations, gets one, and invokes the Gateway's `onInvite()` handler. The Gateway simply extracts the desired telephone number from the `InviteSignalEvent` and uses one of its own lines to place a call to that number. The Gateway also sets up an audio connection between it and CUPS:

```
Gateway in-channel:  dx_play <- networkSource
Gateway out-channel: dx_record -> networkDestination
```

An interesting situation is when a user calls the Gateway and leaves the extension of another user, who turns out to be available only at another telephone. In this case, the AudioConnection will go from the telephone to the Gateway, to the network, in from the network, to the Gateway, and out on the other telephone line. A future optimization would be to catch this happening, and reroute the first call directly to the second.

9 TBD: CDR output

An important function of the control layer is to log changes in control state for potential billing of calls. Typically signaling objects do this by emitting Call Data Records, which include such things as time a call was initiated, who made the call, when was the call terminated and why, etc.

Rather than having each *DesktopSignaling* writing to its own disk file, we would probably choose to have the signaling objects maintain a Stats structure (currently in the `cnrg.itx.datax` package) from which a billing or logging application can pull the desired data.

Here is one suggested list of CDR fields (from the Pulver Report) [1]:

1. Start Time (based on Atomic Clock time)

2. Stop Time

3. CallerID

4. Dial Digits

5. AB Number (?)

6. Origination IP Address
7. Destination IP Address
8. Destination Number
9. Gatekeeper IP Address
10. Bit Rate
11. Codec Selected
12. Network Metric (# of hops)
13. Round Trip Delay
14. QoS relative to what was purchased
15. Reason the Call was Dropped
16. Billing Method for the Call
17. Packet Loss %

10 Conclusion

What can one conclude?

References

[1] Jeff Pulver. Suggested CDR fields, 1998. `cdr@pulver.com`.

[2] H. Schulzrinne and J. Rosenberg. The session initiation protocol: Providing advanced telephony services across the internet. *Bell Labs Technical Journal*, 3(4):144–160, October–December 1998.

The LaTeX source for this report can be found in
/amd/sundial/a/bergmark/public/NSF-telephony-proposal/SignalingInterface/package.tex.