NTNU

Norwegian University of
Science and Technology

Engineering Responsive Mobile Applications for Android from Reusable Building Blocks

Geir Sagberg

Master of Science in Communication Technology
Submission date: June 2011
Supervisor: Peter Herrmann, ITEM
Co-supervisor: Frank Kraemer, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Student: Geir Sagberg

One quality mark of mobile applications is their responsiveness, that is, how a user perceives that the application accepts feedback and reacts to events. Another aspect is how these applications are integrated within the system's application life cycle, so that there is a good balance between what they have to do in the background, the attention they require from the user and their resource consumption.

In this thesis, we want to study how Arctis can contribute to build applications that are responsive and that integrate well with the application life cycle model of Android. As basis, an existing voice communication application should be improved and extended. The focus not only lies on the quality of the resulting application with respect to responsiveness and its integration into the operating system, but also the quality of the specifications and how well they can be reused in other applications.

Assignment given: 17.01.2011

Supervisor: Peter Herrmann, Professor, ITEM

Co-supervisor: Frank Alexander Kraemer, Ph.D., ITEM

Abstract

This report describes the continued design and development of an instant voice communication application for Android, with specific focus on creating a highly responsive, stable application that is intuitive to use and integrates well with the Android environment. Existing building blocks have been redesigned with cleaner layouts and smaller state spaces, and new reusable blocks have been added. Techniques and principles for optimizing an application for responsiveness will be presented, along with specific measures for Android and Arctis. Another goal has been to create the first Arctis application to be released on the Android market.

As a part of the design process, we have researched the development of Android services in Arctis. All service implementation variants have been examined, and the available patterns for communicating between a foreground activity and a background service have been analyzed and compared. The result is a general development methodology for creating a single Android application from two Arctis system models representing a background service and a foreground user interface, with the necessary Arctis modifications included.

Preface

This report documents the results of my work in the course TTM4905 - Network and Services, Master Thesis, during the spring semester of 2011. The thesis is the final part of the Master's degree program in Communication Technology at the Department of Telematics (ITEM), Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Peter Herrmann, and my co-supervisor Frank Alexander Kraemer for invaluable guidance, quick e-mail assistance and regular updates to Arctis.

Geir Sagberg,

Trondheim, 20th June 2011

Contents

List of Figures

List of Tables

Acronyms

AIDL Android Interface Design Language

ALC Activity Life Cycle

ANR Application Not Responding

API Application Programmable Interface

InVoMe Instant Voice Messenger

ESM External State Machine

GUI Graphical User Interface

OS Operating System

RPC Remote Procedure Call

RTP Real-time Transport Protocol

UI User Interface

VM Virtual Machine

XML Extensible Markup Language

XMPP Extensible Messaging and Presence Protocol

Chapter 1

Introduction

Mobile phones and devices have grown drastically in power and versatility over the last decade. Advances in technology give better hardware, which in turn support a wider range of software and applications. The move from device-specific operating systems to common platforms such as Symbian [1], Android [2] and iOS [3] has also laid the ground for "the app revolution": thousands of free and inexpensive lightweight applications available from the online application stores, with more added daily.

For an application to be successful, it needs to feel stable and responsive to users. Applications that crash, or cause the user interface to freeze, are perceived as bad or unfinished software, and will usually provoke low ratings in the app stores. Furthermore, even though an application may not crash or freeze, it might still misbehave and use more than its fair share of the device resources, lowering battery life and causing the rest of the system to appear unresponsive.

The focus of this report is the continued development of the Instant Voice Messenger (InVoMe) application detailed in [4], re-branded as NTNU Instant Voice Messenger. As with InVoMe, the application is developed using the Arctis modeling framework [5], and the new and updated building blocks will be shown and explained.

Special care has been given to producing a highly responsive and well-behaved application. Nearly every existing building block has been redesigned with focus on a cleaner state space, graceful error handling and well-defined behaviour. We have researched principles and techniques for creating a responsive, seamless and intuitive user experience, and applied this knowledge to every aspect of our application.

This is also the first Android application developed in Arctis with interaction between a foreground user interface and a background service. As such, the development of an Android Service in Arctis will be analyzed and generalized for use in other projects.

1.1 Application Overview

NTNU Instant Voice Messenger is a stable and responsive communication application for Android. It is based on the Instant Voice Messenger (InVoMe) application developed in my project thesis [4], improved with a two-part architecture consisting of a foreground user interface and a background service, and extended with new features.

- Enables recording and streaming of voice audio messages between Android terminals
- Uses an existing Google account (GMail) and contact list
- Color-coded presence - available, busy, away or unavailable
- Can receive messages while running in the background
- Messages received when busy are automatically stored for later playback
- Supports text chat and email with any Google Talk contact

The application opens with a login screen, where the user can enter their GMail credentials. After logging in, the user's contacts are shown, with color-coded presence and notifications for voice-compatible contacts and stored messages.

Selecting a contact displays the choices available for that contact, depending on their status. The application can be closed by pressing "Back" on the handset, while the service continues running in the background. The user can log off either by pressing "Menu" -> "Disconnect", or by selecting the service notification in the status bar.

On the Internet

NTNU Instant Voice Messenger is the first application developed in Arctis to be published at the Android Market [6]. A demonstration video can be found at the Arctis page on Vimeo [7]. The source code of the application can be found at the Arctis github page [8], though access is restricted to members of the Arctis group.

1.1.1 Application Use Case Scenarios

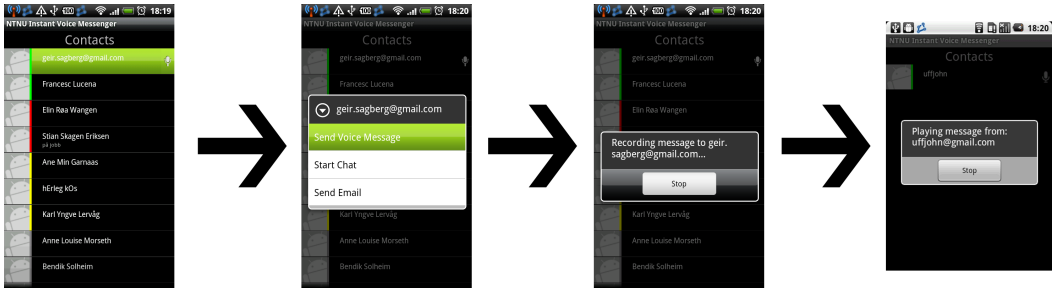
The following scenarios illustrate the different uses of the Messenger application. Figure 1.1 illustrates the scenarios with screenshots, although the names differ from the ones in the scenarios.

Real-Time Voice Messaging John and James are working on a project from two separate locations, both running the Messenger application. John finds a bug, and sends a voice message to James asking him to fix it. James instantly receives the message, fixes the bug, and sends a voice message back to confirm.

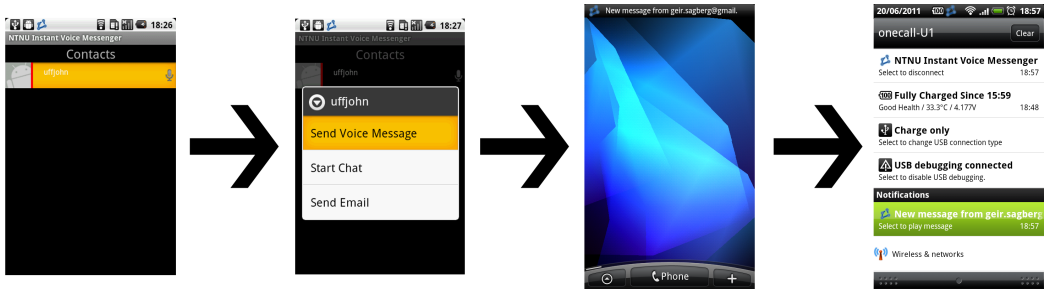
Asynchronous Voice Messaging John is at home and discovers there is no more milk. He sends a voice message to his room mate Mary, asking her to pick up some milk on her way home from work. Mary is currently in a meeting, so the message is stored for later, and a discreet notification is shown on her phone. When she exits the meeting, she plays the voice message from John, and picks up milk on the way home.

Text Chatting John is lonely, and wants to chat with someone. He opens the Messenger application and browses his Google contacts. He finds Peter is online, but not currently using a voice-compatible client. John initiates a text chat, which opens in the Google Talk application for both John and Peter. They chat by text, and John is no longer lonely.

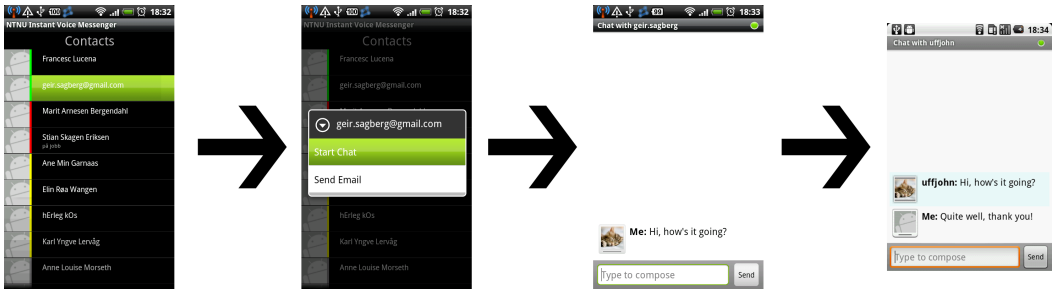
Email John and James are still working on the project, eagerly passing voice messages back and forth. James then has to leave, and logs off the Messenger application. John finds a new bug in James's code. Seeing James is no longer online, John uses the application to start an email, detailing the bug he has found.



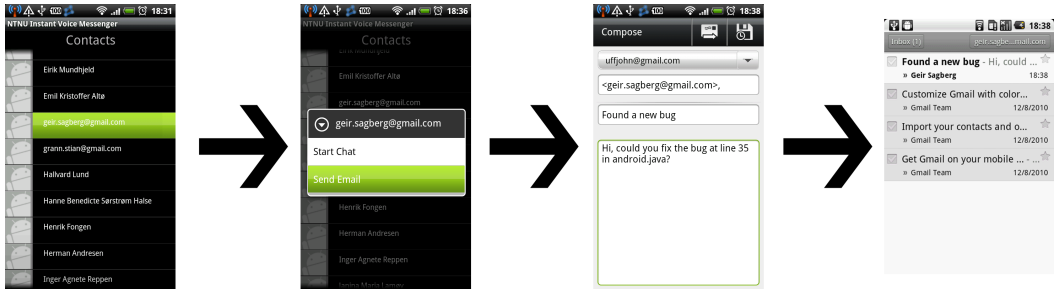
(a) Real-time voice messaging



(b) Asynchronous voice messaging



(c) Text chatting



(d) Email

Figure 1.1: Illustrations of the four Messenger scenarios

The next time James checks his email, he finds the message from John, and fixes the bug.

1.2 Separation of Functionality

The first version, InVoMe, could only receive voice messages while the application activity was running. While the activity could run in the background by pressing "Home", there was no way to see if the application was still running, and it might be closed at any time by Android, to save resources.

To provide a more stable user experience, the application functionality was split in two: a background service containing the XMPP connection to the Google Talk server, the RTP receiver and the audio playback functionality, and a foreground application providing a graphical user interface, audio recording and RTP sender functionality.

The two parts run in separate processes, and communicate by broadcasting Android *Intents*. By default, any application can listen to broadcasted intents, but as we need to protect sensitive information like the user's Google account name and password, the intents contain signature permission restrictions - only applications signed using the same developer certificate can listen for the intents.

In order to run an application created with Arctis as a service, we had to develop a wrapper class that inherits from the *Service* class, and replace the existing wrapper class that inherits from the *Activity* class. In addition, we found that special measures must be taken when the project is generated, to merge the foreground application and the background service into a single Android application. This process will be examined in Chapter 3.

1.3 New Building Blocks Developed

During the development of NTNU Instant Voice Messenger, some blocks from the last version were reused, others were modified, and several new blocks were created. This section will give an overview of the new libraries and blocks used in the application, as well as prototypes and test applications. An overview of the blocks used in the previous version can be found in [4, Ch 1.2], a summary will also be given in 2.3.

Every following block has been analyzed by Arctis, and most blocks have no issues. Where a block has issues, this will be commented in the relevant chapter. External State Machines (ESM) for the most important blocks will be shown in their respective chapters and sections.

Audio Contains blocks for playing and recording audio. A new reusable block, *Play Audio 3*, was added. This block is updated to automatically stop playing when it reaches the end of the input stream.

File Contains the reusable blocks *Read File* and *Write File*, used for reading and writing files to and from the file system.

InVoMe 2 The main library for the application, contains mostly application-specific blocks, but also a few reusable, generic blocks. The most important blocks are:

Contacts UI 2 The redesigned user interface block, responsible for displaying the contact list and updates, and handling input from the user. See Chapter 4.3 for details.

InVoMe Service The system block representing the background service part of the Messenger application. Examined in detail in Chapter 5.

NTNU Instant Voice Messenger The system block modeling the foreground interface part of the Messenger application. Examined in detail in Chapter 4.

Notification 3 A generic block for sending Android notifications.

Play Message Wrapper around *Play Audio 3*, plays back a specific voice message.

Receiver Uses the *RTP Receiver 3* block to listen for incoming voice messages. Also keeps a mapping between user names and IP addresses, for saving messages with their related user name.

Save Message Uses *Write File* to store a specific voice message to SD card.

Sender 2 Uses *Audio RTP Send 2* to send a voice message to a given contact. Also displays a "Sending..." dialog.

Service Controller Generic block used in a system block representing a service, to start the service in either foreground or background mode. See Chapter 3 for details.

XMPP A wrapper around *XMPP Client 5* and *XMPP Gateway*.

XMPP Gateway A redesign of *Voice Support* in InVoMe, translating between XMPP-specific objects and generic objects, for contact and presence updates, etc.

RTP Contains blocks for sending and receiving data over RTP [9]. Two new blocks were added, *RTP Sender 3* and *RTP Receiver 3*. These blocks add "end-of-stream"-recognition, as well as timeout support. *RTP Receiver 3* supports simultaneous reception from several participants, creating a new input stream for each participant. Blocks for signaling over RTCP were also prototyped, but not used in the final application.

Service Contains application prototypes for running an application created with Arctis as an Android Service, and several versions of test applications for different types of foreground-background communication. See Chapter 3 for details.

XMPP Contains generic blocks for connecting to an XMPP [10] server. *XMPP Client 5* has been added, designed for improved stability by eliminating blocking transitions and reducing the number of states from earlier versions.

1.3.1 Additions To Arctis Android Library

The following blocks were created and added to the Arctis Android Library during development:

ALC 5 A new version of the Activity Life Cycle (ALC) block, with a cleaner layout and smaller state space.

Broadcast Receiver 2 A broadcast receiver that can listen for multiple intents instead of just one.

CancelableButtonDialog A one-button dialog that can be dismissed without the user pressing the button.

Progress Dialog 3 A dialog showing an updatable progress bar.

1.3.2 Development Environment

Development was done in Eclipse Classic 3.6.2 (Helios) on Windows 7 and Mac OS X Snow Leopard, using the regularly updated Arctis plug-in alpha (latest version 1.0.0.M0450) and Android Developer Tools plug-in for Eclipse, version 10.0.1. Subclipse v1.6 and Git v1.7.5 were used for version control. Application testing was performed with a HTC Desire and a HTC Hero, both running Android 2.2.

Chapter 2

Background

This chapter will provide a short introduction to relevant technologies and concepts. A basic knowledge of the Android operating system and the Arctis framework is recommended; [2] provides a short introduction to Android, and documentation for Arctis can be found in [11], or at the Arctis homepage [5]. For specific details regarding the implementation of the Android activity life cycle in Arctis, see [12].

We will define the concept of responsiveness and present techniques for improving the responsiveness of an Android application, including Arctis-specific measures. The Android application architecture will be detailed and explained, and the InVoMe application developed in [4] will be introduced.

The RTP and XMPP protocols and the respective APIs used will not be described here; a thorough description can be found in [4, Ch 2].

2.1 Building Responsive Applications

Mobile devices such as smartphones are mostly used "on the go", when one is away from a laptop or desktop computer. The user does not expect to be able to perform the same tasks as on a PC, but the tasks they *do* expect to perform, they want to perform as smoothly and effectively as possible.

A major aspect of the user experience on a mobile device is how responsive the device and application seems. For our purpose, responsiveness can be defined as an application's ability to respond to the user's input in a timely and expected

fashion, without stuttering or freezing. For user input, three important response time limits can be recognized [13, Ch 5.5]:

Instantaneous response: < 100 ms For input reactions to appear instantaneous, they should happen no more than 100 ms after the input event. Any higher delays will be noticeable by the user, and the UI will no longer feel "snappy".

Seamless operation: < 1 second Once an operation takes longer than about a second to complete, the user's flow of thought is interrupted and visual feedback should be provided, e.g. a progress dialog, to keep the user's attention.

Attention limit: < 10 seconds If an operation takes longer than 10 seconds, the user will likely have lost their attention to the task at hand. Visual feedback showing percentage or time remaining should be provided, and the user should be given an option to cancel the operation or switch to a different task.

2.1.1 Techniques for Improving Responsiveness

To improve responsiveness in an Android application, there are a number of measures that can be taken:

Separate Threads for Blocking Operations

In Android, the user interface runs in a single thread, and a common development mistake is to perform heavy operations or network calls directly in this thread. This will block the thread from updating the user interface or responding to user input, and if the UI thread is blocked for more than 5 seconds, the Android OS will kick in and display the "Application Not Responding" (ANR) dialog (Figure 2.1), giving the user an option to force close the application [14].

To prevent an ANR, there are several options, all of which involve moving the blocking calculations and methods to a separate thread or process:

Thread The simplest approach for avoiding blocking the user interface is to create a new *Thread* [15], and handle calculations there. Threads can be given

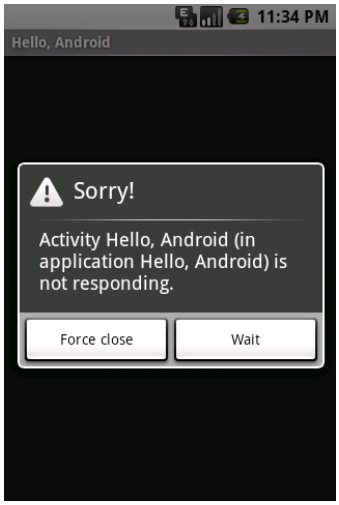


Figure 2.1: "Application Not Responding" dialog from [14]

different priorities according to their importance, e.g. a thread for recording audio should be given higher priority than a thread doing database indexing in the background. Communication with the main thread can be done via static methods and variables, but this can cause concurrency problems (deadlocks, inconsistent data) if not done carefully. To avoid concurrency issues, communication can be done via a *Message* sent to a *Handler* running in the main thread.

AsyncTask Android provides the *AsyncTask* class specifically for asynchronous operations that work on a set of data in the background, producing continuous updates and/or a final result. This class is not used in our application and will not be mentioned further, however an example can be found at [16].

Service A service can run in a separate process from the main activity, and is suited for more complex functionality such as recording or playing audio, handling network connections or performing physics calculations. Note that a service running in the main process uses the main thread, and must create new threads to handle blocking operations.

Asynchronous Screen Updates

When presenting large amounts of data, or data arriving over a network connection, it may be tempting to wait until all the data has been loaded into memory

before displaying it on screen. However, usually the user will want some kind of result as soon as possible; this can be done by handling data loading in a separate thread with updates continually sent to the GUI thread. In this way the user interface will still remain responsive, and information will be available to the user at the earliest possible moment.

In Android only the main thread may directly access the screen element objects, so any updates to the screen must either be wrapped in a *Runnable* block and started with *activity.runOnUiThread(Runnable r)* or *handler.post(Runnable r)*, or sent as data in a *Message* to a *Handler* running in the main thread.

Control Element Feedback

In the real world, when we press a button or handle mechanical equipment, we can immediately feel that something happens; buttons are physically depressed, levers are pulled etc. When clicking or touching icons and buttons on a screen, if we do not get some kind of response, it is hard to know whether our actions actually had an effect. By adding simple feedback, such as blinking a button when pressed or vibrating the device, we acknowledge the user's input, and fulfill their immediate expectations. This may even be enough to keep the user interface feeling responsive even if the subsequent reaction takes longer than 100 ms.

In Android, most controls have visual feedback built-in, and haptic vibration feedback can be added with the XML attribute *android:hapticFeedbackEnabled* and calling the method *aView.performHapticFeedback()*.

Graceful Handling of Runtime Errors

Coding errors can cause infinite loops or deadlocks, eventually bringing about the ANR dialog in Figure 2.1. However, coding errors can also cause runtime exceptions, and if not caught they will prompt an "Application stopped unexpectedly" dialog (Figure 2.2), forcing the user to close the application.

Runtime errors can be hard to anticipate and debug, but the API documentation may give an overview of possible thrown exceptions. If it is known that a method call may throw a runtime exception, it is better to wrap the call in a try/catch block, and present an error message to the user in a graceful way, ideally allowing continued operation of the application rather than forcing a close.

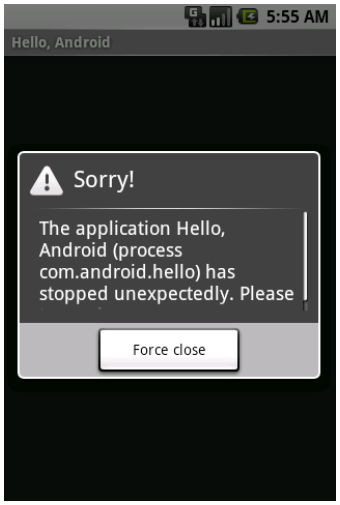


Figure 2.2: "Application stopped unexpectedly" dialog from [14]

Time and Percentage Feedback of Long-Running Operations

When an operation runs for an extended amount of time, if no indication of progress is shown, the user might get the impression that the application is stuck. For operations where the size of the data is known, percentage can be easily calculated. Android provides a *ProgressDialog* [17] with a bar that is updated as the operation proceeds. Time remaining can be calculated by timing the operation and extrapolating, adjusting for variable speed if need be.

For operations of unknown size or time, *ProgressDialog* can also be shown as an indeterminate animated symbol, as in Figure 2.3. If the operation has multiple stages, the dialog can be updated with descriptions, e.g. "*connecting...*", "*authenticating...*", "*loading contacts...*". In this way, the user is reassured that something is actually happening, and will hopefully endure the waiting.

Cancelable Operations

If an operation takes too long time, the user might decide that it is not worth it, and may want to abort the operation. All Android phones have a "Back"-button, which always should cancel the current activity and return the user to their previous activity. If the user attempts to cancel a long-running operation and nothing happens, the application will seem unresponsive. To correctly handle cancella-

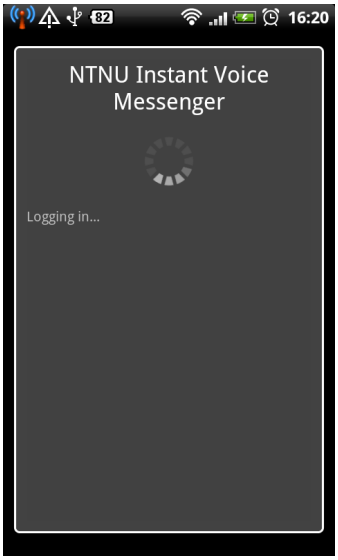


Figure 2.3: Indeterminate progress dialog

tions, the dialog can be made cancelable and assigned an *OnCancelListener*, with logic for stopping the operation gracefully.

2.1.2 Responsiveness in Arctis

Arctis uses a background thread for running the runtime scheduler, and will normally not block the UI thread. However, building blocks can use the method *getHandler().post(aRunnable)* to run code on the main thread, e.g. for displaying dialogs. In this case, the same care must be taken as with regular code running on the main thread.

Since every state machine transition in Arctis is handled by a single runtime scheduler thread, blocking operations should be wrapped in runnables and run in separate threads, to avoid lost transitions and blocking functionality.

Runtime errors in Arctis are caught by the runtime scheduler and displayed in the Arctis Logger. However, to avoid disrupting the user with error notifications, building blocks should anticipate runtime errors when possible and provide output pins with *Exception* objects, so runtime errors may be handled as part of the program flow and subjected to automatic analysis.

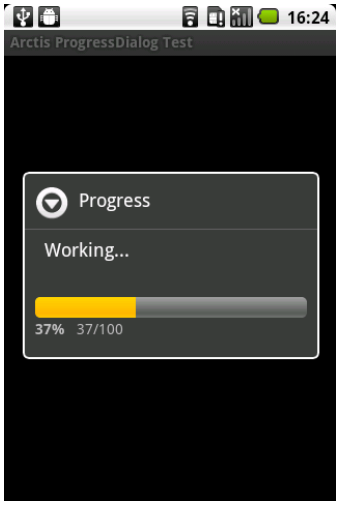


Figure 2.4: *Progress Dialog 3* in action

For progress feedback, the Arctis Android Library contains progress dialog blocks usable in any Android application. The blocks *Progress Dialog 1* and *Progress Dialog 2* show an indeterminate spinning progress wheel. *Progress Dialog 3* (Figure 2.4) was developed during this project but not used in the application. It shows a determinate progress bar that can be updated via a *setProgress*-pin.

Creating cancelable building blocks is as simple as implementing an *OnCancelListener* for the UI element in question, and sending a signal to the Arctis model so a *cancel*-pin can be triggered. Every UI block used in our Messenger application has been made cancelable.

2.2 The Android Application Framework

To create a responsive and well-behaved Android application, a certain understanding of the Android operating system and application structure is needed. We will now provide an overview of the components of an application, with a deeper look into services and communication between a foreground activity and a background service. Lastly, we will present best practices for seamless integration with the Android environment.

2.2.1 Application Components

Android applications consist of four different components: *Activities*, *Services*, *BroadcastReceivers* and *ContentResolvers*. We will briefly explain all of them except content resolvers, which are not used in our application. A more thorough description of each component can be found in [18].

Activity Every screen the user sees is part of an activity. Activities have a set of *Views* (text, images, buttons etc.) and usually take some form of input from the user. Activities may start other activities, services or broadcast receivers. Not to be confused with UML activities such as the models used in Arctis. The latter will, to avoid confusion, be referred to as Arctis applications, building blocks, or simply models.

Service Services run in the background, hidden from the user. They can be used to perform operations in the background (playing music, indexing files etc.) or can be bound to perform specific method calls.

BroadcastReceiver Broadcast receivers listen for specific events (*Intents*), either from the system, e.g. when a headset is plugged in, or custom events sent from applications. Like activities and services, broadcast receivers can be declared in the *AndroidManifest.xml* file [19], but they may also be registered from inside a service or an activity.

All components in an application run by default in a single Linux process with its own private memory space. However, a component can be put in a separate process by specifying a process name in *AndroidManifest.xml*. The component will then run in a separate Dalvik virtual machine and can not share memory (static variables, etc.) with the other components [18].

In addition to the main components above, our service implementation will make use of the *Notification* and *Handler* classes:

Notification When an application wants to notify the user of something without interrupting the current activity, it can use the *Notification* class to show a message in the status bar, optionally with an action that is performed when the notification is selected.

Handler The *Handler* class can be used for safe communication between threads (and processes, if wrapped in a *Messenger*). Any class with a reference to a

handler can send messages to it, optionally with a *Bundle* of serializable or parcelable data [20]. The messages will then be received in the context and thread of the handler.

2.2.2 Service Implementation and Communication

A service can be implemented in several different fashions. It can be started once and run indefinitely, or bound to an interface and automatically destroyed when no longer needed. It can run locally - in the same process as the main activity, or remotely - in a separate process. It can run in the background, hidden to the user, or in the foreground with a compulsory notification displayed in the status bar.

Started vs. Bound

A service started with *context.startService()* ("started service") will run until it receives a *finish()* call, or until it is killed for memory. Several calls to *startService()* will not create multiple instances, but the service's *onStartCommand()*-method is called every time. This can be used to pass data for background operations. Started services are suitable for operations running independently of a user interface, or operations that should continue after a main activity is closed.

A service started with *bindService()* ("bound service") will return an *IBinder* interface for remote method calls, and will only keep running as long there are contexts bound to it. A service may be both started and bound, in which case it will continue running even after the binding context is ended. To disallow the binding of a service, the *bindService()*-method can simply return *null*. Bound services are suited for client-server type interactions, and for services needed only in the context of a user interface, e.g. a rendering engine.

Local vs. Remote

Local services bound with *bindService()* can create an extension of the *Binder* class, e.g. *MyBinder*, so that the binding activity may cast the *IBinder* interface to *MyBinder* and use its methods directly. A local service implementation is simple and uses less memory than a remote service, as only one process is needed.

Bound services running in a separate process must either use the *Messenger* class with a *Handler* to provide a message passing interface, or describe a full RPC interface using the Android Interface Definition Language (AIDL) [21]. AIDL should be used if multi-threaded operation is needed, but requires explicit thread handling and may be overly complex for simple services.

When a remote service is started with *startService()*, communication can only be done by passing data with intents, either with repeated calls to *startService()* or by registering broadcast listeners at service and activity, and broadcasting intents back and forth. When intents are broadcast, by default any receiver can listen for them; if sensitive data is passed, the intents must be imprinted with signature permissions, restricting reception to applications signed with the same developer certificate.

Local services can also communicate by using singletons or accessing an activity's static variables, and vice versa. However, this will introduce global state to the application and make testing harder. In a multi-threaded application, extra care must be taken to make the static methods thread-safe, e.g. by not allowing the method to change an object that may be accessed by several threads at once.

Background vs. Foreground

A service runs by default as a background service, and may be killed by the system if memory is scarce. If it is important to keep the service running at all times, it can be flagged as a foreground service, and a notification icon must be displayed in the status bar to show that the service is running. A foreground service will not be killed unless under extreme memory conditions.

2.2.3 Best Practices for Seamlessness

To provide a seamless user experience, certain guidelines should be followed when creating applications [22]:

Avoid Popups Although it is possible to display dialogs and start activities from background threads or services, doing so may interrupt the user's current task. Instead one may use a *Notification* which will be shown discreetly in the status bar, and may contain an *Intent* for starting an activity. In this way, the user may decide whether to leave their current task.

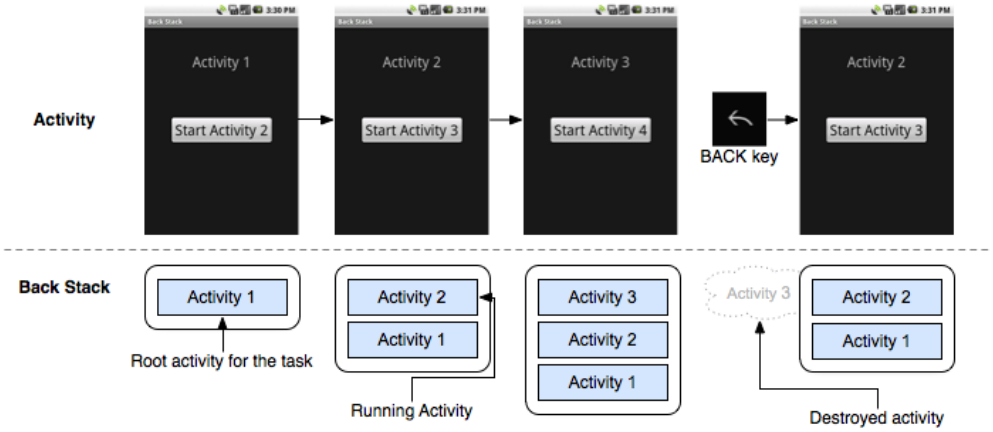


Figure 2.5: The "back stack". Opening new activities puts them on the stack, pressing "Back" pops them. Taken from [23].

Intuitive Navigation When the user makes a selection that causes a new dialog or activity to appear, the view is added to the stack of the activities the user has visited in the current task. By default, pressing "Back" causes the current activity to pop from the stack, bringing back the previous activity (Figure 2.5). However, it is possible to override this behaviour by remapping the back button or displaying uncancelable dialogs. While this may be suitable in certain situations (e.g. using the back button as a button in a game or displaying a dialog during an operation that must not be canceled), as a rule it is better to preserve the default behaviour to provide a seamless and intuitive user experience.

Persistent State Android supports multitasking, but in contrast to PCs, mobile devices have limited memory and no swap support [24]. Therefore, when the current activity needs more memory, background applications may be shut down by the system. However, the user expects to be able to switch to a previous task and continue where they left off; to this end, the *Activity* class provides the methods *onSaveInstanceState()* and *onRestoreInstanceState()*. The methods are called when an activity is about to be stopped or restored, respectively, and can be overridden to store the current state of the application.

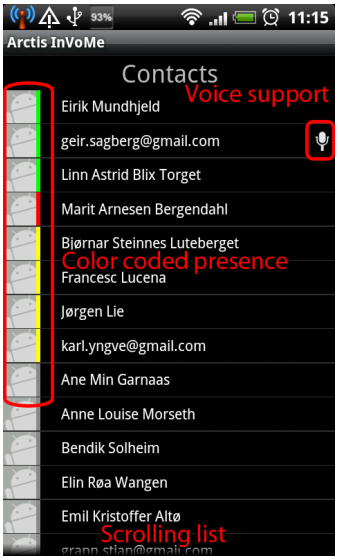


Figure 2.6: User interface of InVoMe

2.3 Instant Voice Messenger (Version 1)

Instant Voice Messenger (InVoMe) is the result of my project thesis in the fall semester of 2010 [4]. It is a fully functional voice message application for Android, designed with Arctis using reusable building blocks. The application uses the XMPP protocol [10] for connecting to the Google Talk servers with an existing Google account, and supports streaming of voice messages over RTP [9] between InVoMe clients.

The application model is shown in Figure 2.7. It consists of the following main blocks:

Contacts UI The main user interface, shown in Figure 2.6. Displays a list of the user's Google Talk contacts, with a microphone icon denoting voice message compatible contacts.

XMPP Client 4 Handles the XMPP connection to the Google Talk servers, and sends and receives roster and presence updates. Also supports text chat, but this functionality is not used in InVoMe.

Voice Support Translates between XMPP-specific objects and the generic objects used in *Contacts UI*.

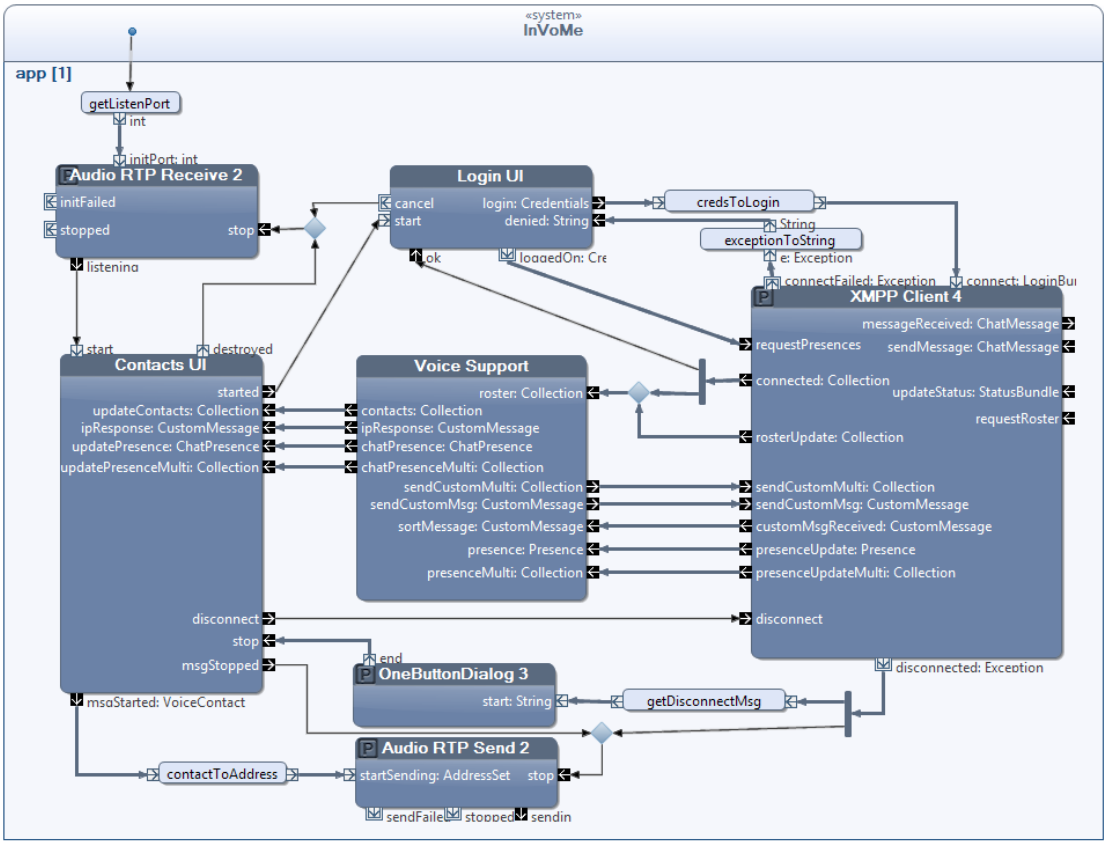


Figure 2.7: Internal behaviour of InVoMe

Audio RTP Send & Receive 2 These blocks send and receive voice messages over RTP, respectively.

Two blocks from the Arctis Android Library, *Login UI* and *OneButtonDialog 3* are also used.

2.3.1 Main Version Differences

Features All the features from the first version are kept in the current version. In addition, when a voice message arrives when the client is busy - either already receiving a message, or running in the background - it is cached by the application, and may be played back at a later time. Chat and email functionality has been added via *Intents*, launching the appropriate applications if installed on the device, e.g. GMail or Google Talk.

Service InVoMe is modeled as a single system block, and runs on a single Arctis runtime, in a single Android process. This architecture was chosen because of its simplicity; no extra customization of Arctis is needed to produce the resulting application. However, to conserve memory, the application may be automatically shut down by the system as soon as it is no longer in the foreground. The current version alleviates this by separating non-GUI functionality into a service, which is given higher priority than background activities.

User Interface In InVoMe the login credentials have to be re-entered every time; in the new version, the username and password can be saved for faster startup. The contact list has been updated with wider rows, for easier selection. A notification displaying number of stored messages is shown for each contact. Selecting a contact no longer immediately starts a voice message, but instead opens a dialog with options relevant for the state of the contact.

Performance and Concurrency The previous version suffered from a 10 second delay when sending a voice message. We traced the delay to a concurrency problem in the *jlibrtp* API [25], specifically *RTPReceiverThread.java*: When attempting to get the socket address of the first *DatagramPacket* [26] arriving, the thread would hang for about 10 seconds. We eliminated the delay by replacing the following line:

```
part = new Participant((InetSocketAddress) packet.getSocketAddress(),
nullSocket, pkt.getSsrc());
```

with:

```
InetSocketAddress rtpSocket = new
InetSocketAddress(packet.getAddress().getHostAddress(),
packet.getPort());
part = new Participant(rtpSocket, nullSocket, pkt.getSsrc());
```

The delay is likely due to a deadlock somewhere, as all the methods of the *DatagramPacket* class are marked as *synchronized*, i.e. only one thread can access the method at once. We have registered the bug and the suggested fix at the *jlibrtp* SourceForge page [25].

Presence Updates In the previous version, presence updates were sometimes lost when they arrived before the respective contact update; in the new version, presence updates are stored and applied when the matching contact update arrives.

Chapter 3

Android Services in Arctis

In this chapter we will examine how an Arctis application can be implemented as an Android service, and how communication between a service and an activity can be performed. The available communication models will be analyzed and compared, test applications created during development will be shown, and a general development methodology for creating Android Services using Arctis will be presented.

As a result of our experiments we have discovered that when merging two Arctis applications - a foreground activity and a background service - the service must run as a separate process to avoid Arctis runtime collisions. For our Messenger application we have chosen to communicate by passing *Intents* to *BroadcastReceivers*, as this was both easy to implement and visualize, and fits well with the Android application environment.

3.1 From Activity to Service

When an Arctis application is implemented, state machine code is generated and a wrapper activity, *Start,* is set as the Android launch activity. When the application is started on an Android device, the *Start* activity creates the runtime scheduler that processes the generated state machine code.

We wanted to run the application as a service instead, so we decided to modify the implementation procedure and create some prototypes for testing. As a first step we wanted to create a proof of concept that a compiled Arctis application can run as a service instead of an activity.

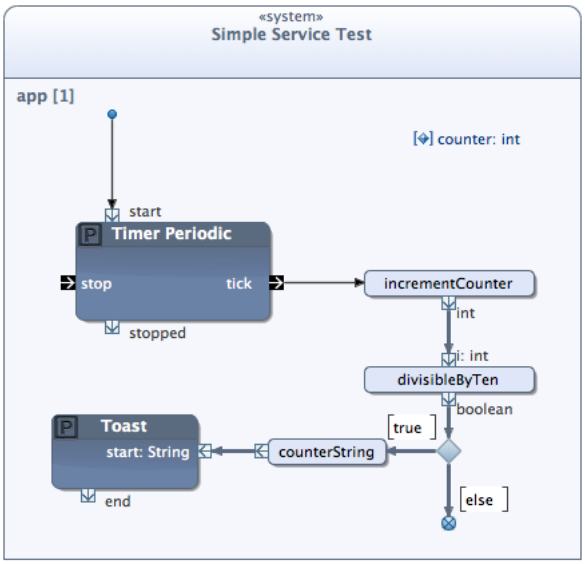


Figure 3.1: *Simple Service Test* application

The *Start* wrapper runs as an activity because it inherits from *GeneralStartActivity3*, which inherits from *Activity* and is part of the Arctis runtime package. We created a new class, *GeneralStartService*, which inherits from *Service* instead.

GeneralStartService adapts the functionality of *GeneralStartActivity3* to the life cycle of a service, starting the scheduler in *onStartCommand()* rather than *onCreate()*. Setup of the static Arctis Logger was removed to avoid collisions when running the service locally with another Arctis application (as we later discovered, only remote services are compatible with Arctis, so the Android Logger may be re-added at a later date). We later added functionality for running the service in foreground mode, this will be covered in Section 3.2.4.

To test the service implementation we created a simple application called *Simple Service Test*, shown in Figure 3.1. The application starts a timer that increments a counter every second. Every tenth second, a toast is shown with the current counter value.

The code was generated, *Start* was edited to inherit from *GeneralStartService*, and the application was tested and found to be working.

3.2 Communication Models and Prototypes

Once we had a working generated service, the next step was to start a service from within another Arctis application, and select a way to communicate between activity and service. Table 3.1 summarizes the communication models from Chapter 2.2.2, showing the service implementations they apply to, with advantages and disadvantages.

3.2.1 Static Methods and Arctis Signals

For our first test application, we wanted to examine the possibility of communicating between foreground activity and background service by using Arctis signals, as an extension of the static method communication model. The idea was to start the service locally with *startService()*, exchange block IDs, and send signals with *AbstractRuntime.getRuntime(blockID).sendToBlock(blockID, signal, data)*.

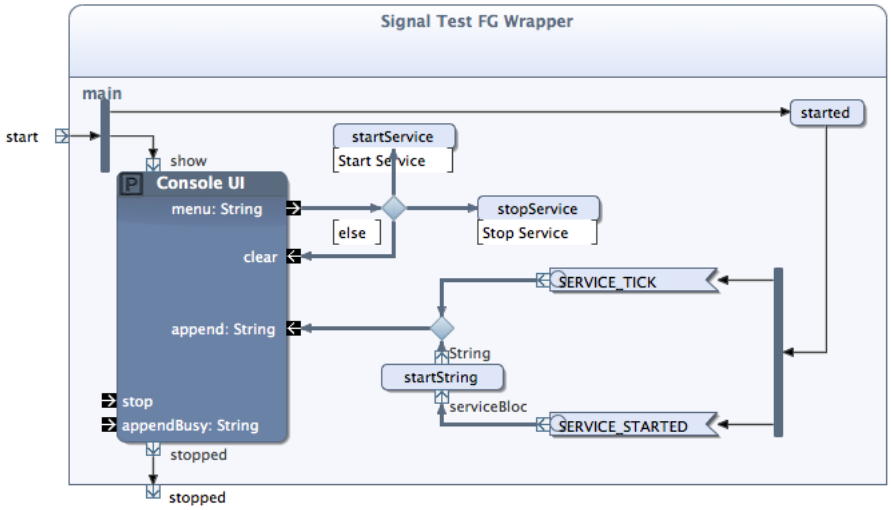
The service would send a signal containing a random message to the foreground activity every second, and the foreground activity would print the message to screen. The models for the foreground and background can be seen in Figure 3.2, wrapped in building blocks so they can use *AndroidBlock*'s method *getContext()* and access the *blockID* field.

In order to run the application, a few extra steps were needed: merging the two generated Arctis applications, editing the service *Start* class to inherit from *GeneralStartService* and adding it to the Android manifest. The merging was done by simply copying all code from the background project to the foreground project, without overwriting. Our background application did not have any extra tags or other resources, else these would have to be added manually in the foreground project Android manifest.

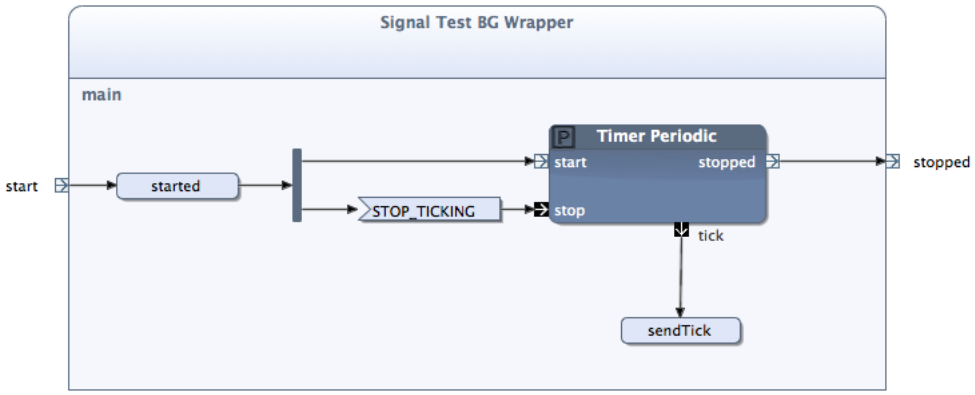
When we ran the application, we found signals were being delivered to the wrong state machines. After some debugging, we found the problem was due to both runtimes using the same memory space. Arctis uses several static fields and methods, specifically a static *Hashtable* in *AbstractRuntime*, *instances*. The table *instances* holds mappings from every block's *blockID* to the runtime it belongs to. However, each block ID is only unique per Arctis application, so when two runtimes each register mappings from their blocks to their runtime, the latter mappings will overwrite the first ones. This caused our background service runtime to receive signals meant for the foreground runtime, invoking *"UNKNOWN TRIGGER"*-errors.

Table 3.1: Comparison of activity - service communication models, summarized from Chapter 2.2.2

Communication model	**Service prerequisites**	**Advantages**	**Disadvantages**
Static methods and variables, singletons	- Local only - Started or bound	- Easy to implement	- Introduces global state
Binder-based interface	- Local only - Started or bound	- Object-oriented	- None
Intents passed with *startService()*	- Local or remote - Started only	- Easy to implement	- One-way communication only - Data must be serializable or parcelable
Broadcast intents to *BroadcastReceivers*	- Local or remote - Started or bound	- Allows two-way communication	- Requires signature permissions to be secure - Data must be serializable or parcelable
Messenger-based interface	- Local or remote - Bound only	- Less complex than AIDL	- Unsuitable for multi-threaded services - Data must be serializable or parcelable
AIDL files	- Local or remote - Bound only	- Supports multi-threaded services - Provides direct method RPC interface	- Very complex - Needs explicit thread handling to be safe - Data must be parcelable



(a) Foreground activity, uses *Console UI* to print messages from the background service



(b) Background service, sends a random message to the foreground activity every second

Figure 3.2: Internal behaviour of *Signal Test* application

3.2.2 Broadcasting Intents

It was clear that we had to use a communication model that supported remote services. This narrowed the choice down to four options: passing intents with *startService()*, broadcasting intents, using a *Messenger* interface or writing AIDL files.

Passing intents with *startService()* is very easy to implement, but only works from activity to service. Also, there is no way to access the received intent from within an Arctis model, without modifying Arctis further. However, by using broadcast receivers, we could pass intents back and forth between service and foreground application, and this seemed a simpler solution than using a *Messenger* or AIDL files.

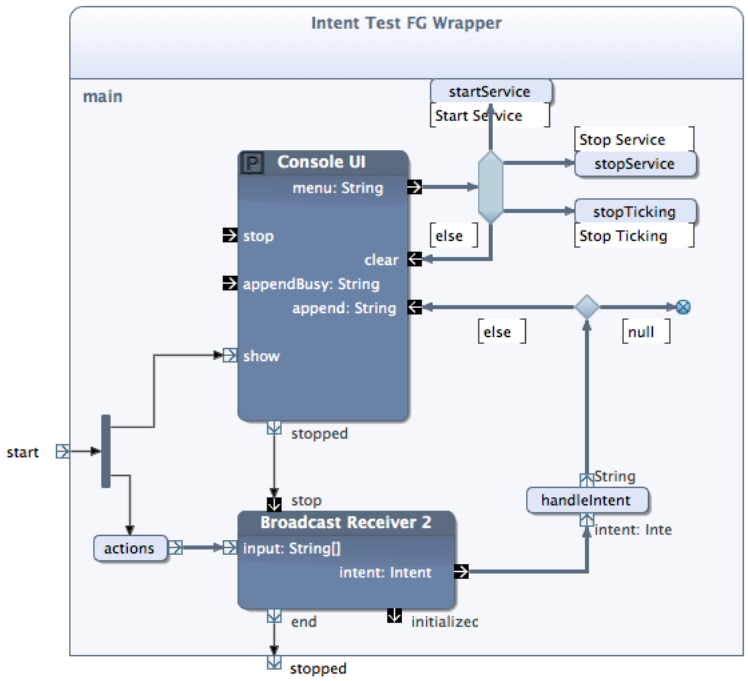
We created a new application, *Intent Test*, which can be seen in Figure 3.3. The Arctis Android Library already contained a *Broadcast Receiver* block, but it only listened for intents matching a single action, so we created *Broadcast Receiver 2*, which takes in an array of action strings. The block is used in *Intent Test FG Wrapper*, while in *Intent Test BG Wrapper* a broadcast receiver is created directly in code, in the *started()*-operation. Creating the receiver directly in code makes it harder to visualize the program flow, but makes it possible to unregister the receiver from code, which may be suitable in models where the event flow is too complicated to easily stop a receiver block.

In order to avoid the shared memory space problem encountered earlier, we added the XML attribute *android:process=":service*" to the service in *AndroidManifest.xml*. This makes the service run in a separate process with its own memory space. The process name is arbitrary, but the ":" appends the name to the package declared in the manifest, and must be included to make the service private.

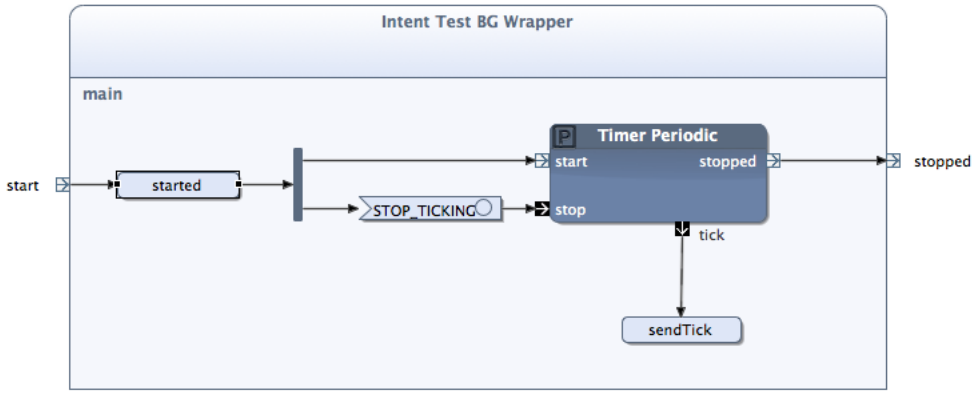
The application was started and found to be working, displaying incoming messages on the screen as they arrived from the service (Figure 3.4). Because of the ease of implementation of this solution, and because it could be done using a building block from the Arctis Android Library, we selected this communication model for use in our Voice Messenger application.

3.2.3 Bound Services and Arctis

The *GeneralStartService* class we have created returns *null* in its *onBind()*-method, and can not be used for bound services. However, while the complexity of full



(a) Foreground activity, uses *Console UI* to print messages from the background service



(b) Background service, sends a random message to the foreground activity every second

Figure 3.3: Internal behaviour of *Intent Test* application

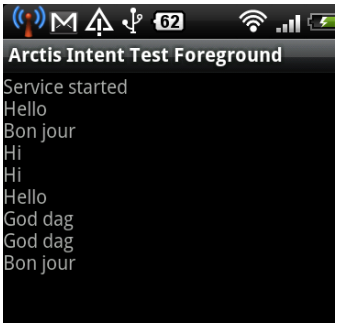


Figure 3.4: *Intent Test* screenshot

AIDL files would prove a difficult implementation challenge in Arctis, using a *Messenger*-based bound service with Arctis should only require a few modifications:

- A new block must be created, *Service Messenger*, which contains a *Messenger* implementation that passes all received messages out through a *receivedMessage*-pin.
  - The messenger must be added to Arctis' runtime context with *AbstractRuntime.getRuntime(blockID).addContext("Messenger", messenger).*
- *GeneralStartService* must be modified in the following ways:
  - An instance of the *Messenger* class must be created, containing a *Handler* which passes messages on to the messenger registered in the runtime context.
  - The method *onBind()* must return the messenger as an *IBinder*, by calling *messenger.getBinder().*
- When the foreground application binds the service, it must cast the received *IBinder* to a *Messenger* in order to send messages.
  - If the foreground wants a reply to a message, it must implement its own messenger and pass it as the field *message.replyTo.*

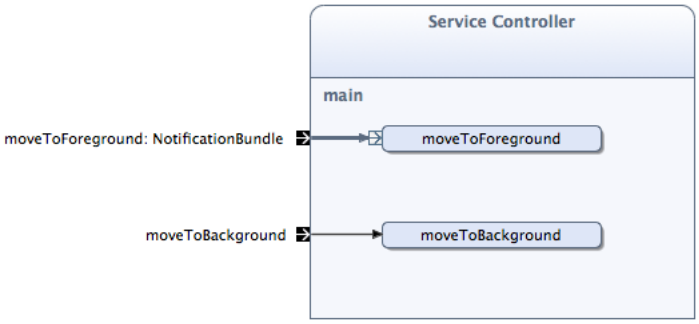


Figure 3.5: *Service Controller*

3.2.4 Running the Service in Foreground Mode

The numerous types of service running on Android can be split in two distinct groups: services that should be hidden to the user, e.g. synchronization services and data processing, and services that the user would want to be aware of, e.g. music players and instant messaging applications. Our application belongs to the latter group, and we therefore decided to run its service in foreground mode, with a notification displayed in the status bar.

In order to run our service in foreground mode, we needed to make some changes to our *GeneralStartService* class. We created an interface, *ForegroundController*, with the methods *moveToForeground()* and *moveToBackground()*. *GeneralStartService* was made to implement this interface by using a code example from the Android API *startForeground()* documentation [27], and to register itself in the Arctis runtime static context, with *runtime.setContext("ServiceController", (ForegroundController) this).*

Then we created a new building block, *ServiceController*, with input pins corresponding to the two methods. The block is very simple, and can be seen in Figure 3.5. Every information needed to create a notification is sent as a *NotificationBundle* to the *moveToForeground*-pin, and passed on to the *ForegroundController* interface registered in the runtime context. The *ServiceController* block can be seen in use in Chapter 5.

3.3 Service Development Summary

We will now summarize the steps needed to create an Android application with a foreground interface and a background service, both modeled as separate system blocks.

1. Create and design both models.

2. Foreground model must start the service with *startService(intent)*. The intent can be created in two ways:

   (a) By name: *Intent intent = new Intent(getContext(), Class.forName("<package of Service>.Start"));*

   (b) By action: *Intent intent = new Intent("<actionString>");*

3. In the foreground model overview:

   (a) Under "Extra Tags", add an *"android_service"* with the value *"<package of Service>.Start"*.

   (b) Under "Required Resources", add any resources used by the background model.

4. Implement both models.

5. Copy classes and resources from generated background project to foreground project.

6. Edit *<package of Service>.Start.java* to extend *GeneralStartService* instead of *GeneralStartActivity3*.

7. To avoid Arctis runtime conflicts, service must run in its own process. Add *android:process=":<any name>"* to the service in *AndroidManifest.xml*.

8. If service is started by action, add a new Intent Filter to the service in *AndroidManifest.xml*, with the same action used in step 2b.

3.3.1 Automation of Service Creation and Merging

Previously, *GeneralStartService* had to be copied over manually, but as of Arctis version 1.0.0.M0450, the *GeneralStartService* class has been added as part of the Arctis runtime package. In order to further automate the service creation and merging process, we propose the following measures:

- When an Arctis model is implemented as a service, the generated *Start.java* should extend *GeneralStartService* automatically.
- A new choice could be added to the generation process: "Add as Service to existing project". The generated code for the service should be automatically merged with the code of the existing Android project, and the Android manifest should be modified as in step 7 above.

Chapter 4

Foreground Application

This chapter will describe the user interface part of the NTNU Instant Voice Messenger application. We will explain the choices made when splitting the application functionality, the building blocks used will be explained in detail, and the program flow will be described. Finally, optimizations for improved responsiveness and seamlessness will be listed, and the state space and code size of this version and the previous version will be compared.

4.1 Separation of Functionality

The previous version, InVoMe, was a single monolithic application, containing functionality for both audio recording and playback, RTP transmission and the XMPP connection, in addition to a user interface. In order to receive messages also when the application was not in the foreground, we needed to split the application in two, a normal Android activity running in the foreground, and a service running in the background. However, before creating the two application parts, we needed to decide which functionality should go where.

XMPP The whole point of creating a service was to maintain a connection to the XMPP server so messages can be received, so naturally the XMPP functionality must be implemented in the service.

RTP and Audio As with XMPP, the reception of messages over RTP is a crucial part of the service, and audio playback is also kept with the service in

order to play received messages via the status bar notification, without starting the main activity. However, the sending of messages will only occur when the user is actively using the application, so we decided to implement audio recording and RTP sending as part of the foreground application.

User Interface Naturally the contact list had to be a part of the foreground activity, and any alert dialogs were also presented only when the activity was running and visible to the user, to avoid undue interruptions. For notifying the user of new messages when only the service is running, we decided to use notifications, because of their non-intrusive behaviour.

4.2 Main Application Model Overview

We wanted to design the models to be easy to read and understand, so we decided on a few simple rules:

- All service-related functionality is wrapped into a single block, *To Service*, with an equivalent *To GUI* block in the service model (Chapter 5).
- The earlier a pin or block is encountered in the event flow, the closer to the top it should be placed. This should make it easier to follow the natural flow of the application.
- To maintain model readability, event flows should have a minimum number of joints and avoid crossing whenever possible.

The resulting model can be seen in Figure 4.1. Compared to the model of InVoMe shown in Figure 2.7, this model is much cleaner and hopefully easier to understand.

The application starts in the top left corner, immediately starting *Contacts UI 2*. When *Contacts UI 2* is active, it sends a signal to the *start*-pin of *To Service*. If the service is already running, nothing happens (a signal is sent through *startedEarlier*, which is connected to nothing), and contact updates will start arriving momentarily.

If the service was not running earlier, *startedNow* fires and *Login UI* is started. The user enters their Google credentials, and these are sent to the service for

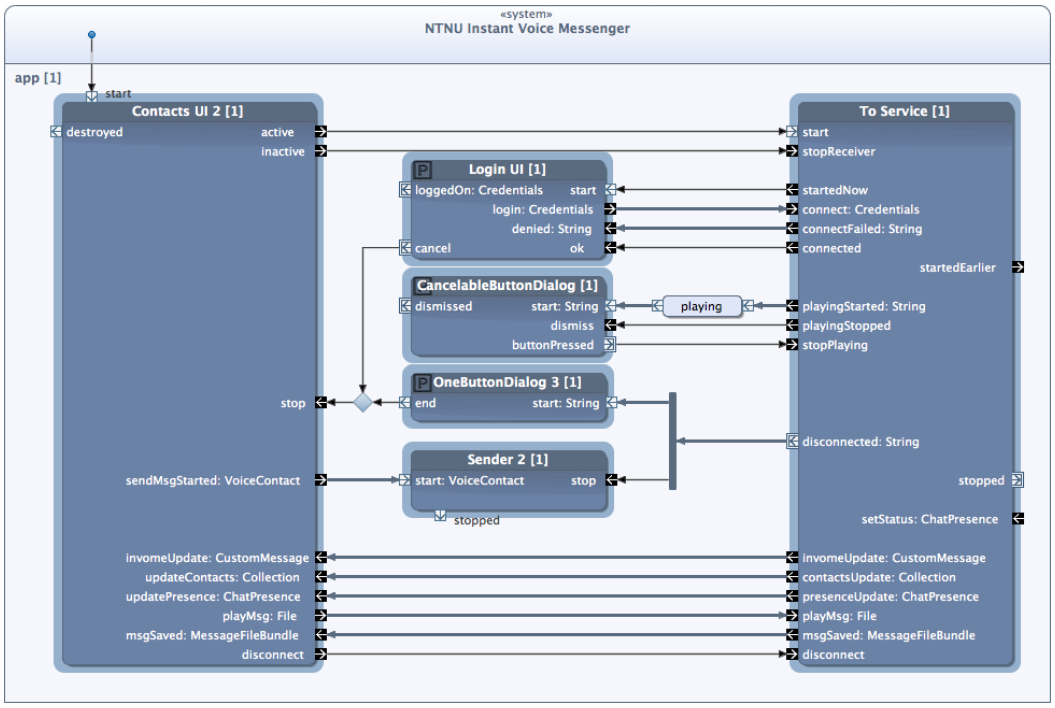


Figure 4.1: Internal behaviour of the user interface part of *NTNU Instant Voice Messenger*

verification via *connect* on *To Service*. Successful authentication fires *connected*, *Login UI* is closed and the user interface is ready for contact updates.

When a voice message arrives, *To Service* fires *playingStarted* with the name of the sender. A dialog box opens, and stays open until the message is finished or the user stops the message by pressing "Stop".

When the user has selected to send a voice message, *Contacts UI 2* fires *sendMsgStarted* with the *VoiceContact* containing the receiving contact. Sender 2 opens a "Sending..."-dialog, and starts recording audio and sending it over an RTP stream to the IP address in the *VoiceContact*. When the user presses "Stop", the message stops.

If the user has selected a previously recorded message for playback, *Contacts UI 2* fires *playMsg* with the specified file, and it is sent to the service for playback, which will again prompt a *playingStarted* from *To Service*.

Three kinds of updates are received from the service:

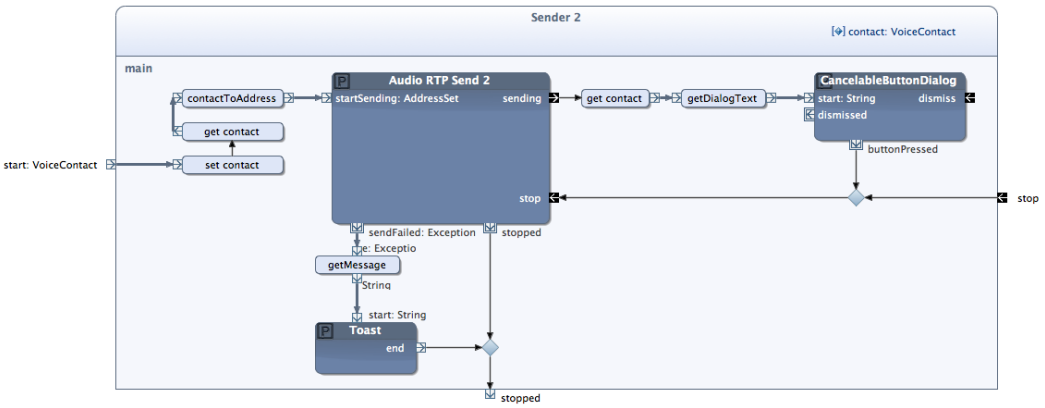
invomeUpdate fires with a *CustomMessage*, containing either an *IP_REQUEST* from another voice compatible client, or an *IP_RESPONSE* in response to a request sent from this client. In both cases, the *CustomMessage* contains the IP address of the other contact, which is stored in the contact list, and a microphone icon is added to the contact.

contactsUpdate fires with a *Collection* of *VoiceContacts*, representing the entire user's contact list. The user interface is updated, and any pending *presenceUpdates* are applied.

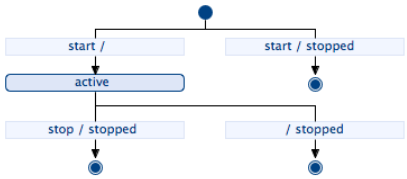
presenceUpdate fires with a *ChatPresence* object, signaling that a user has changed their presence status, e.g. logged off or busy. When the application first starts, every contact's presence is sent, coincidentally with the contact list in *contactsUpdate*. It may happen that the *presenceUpdate* arrives before the contact list. In this case, the presence is stored and applied as soon as the contact list arrives.

When the activity is placed in the background, *Contacts UI 2* will fire *inactive*, which will notify the service to store incoming messages rather than play them, and unregister the broadcast receiver to avoid memory leaks.

When the user disconnects, or the service connection is lost, *To Service* fires *disconnected* with an optional error string. A *OneButtonDialog 3* block shows a



(a) Internal behaviour



(b) External State Machine

Figure 4.2: *Sender 2* behaviour and ESM

"Service Disconnected"-message, with the error message if the user did not disconnect willingly. *Sender 2* is also stopped if it is running.

4.2.1 Sender 2

Sender 2 uses *Audio RTP Sender 2* developed in [4] to record and send a voice message. The internal behaviour and the ESM of the block can be seen in Figure 4.2.

The block is started with a *VoiceContact*, the IP address is extracted and a message is started. If for some reason the message cannot be sent, e.g. if the audio recording device is busy, the block immediately stops. Else, a dialog is displayed with the message "Recording message to <contact>...". The dialog contains a "Stop" button, which stops the message and the block.

While *Audio RTP Sender 2* is mostly unchanged, we have replaced the block *RTP*

Sender 2 with *RTP Sender 3*, which sends a special signal when the end of stream is reached. This signal is recognized by *RTP Receiver 3*, which can then close its own stream.

4.3 Contacts UI 2

The newest version of the contact list user interface has been given a cleaner design compared to its predecessor described in [4, Ch 6.2]. Figure 4.3 shows the internal behaviour and ESM of the block. Only one UI block is used now, simplifying state space considerably.

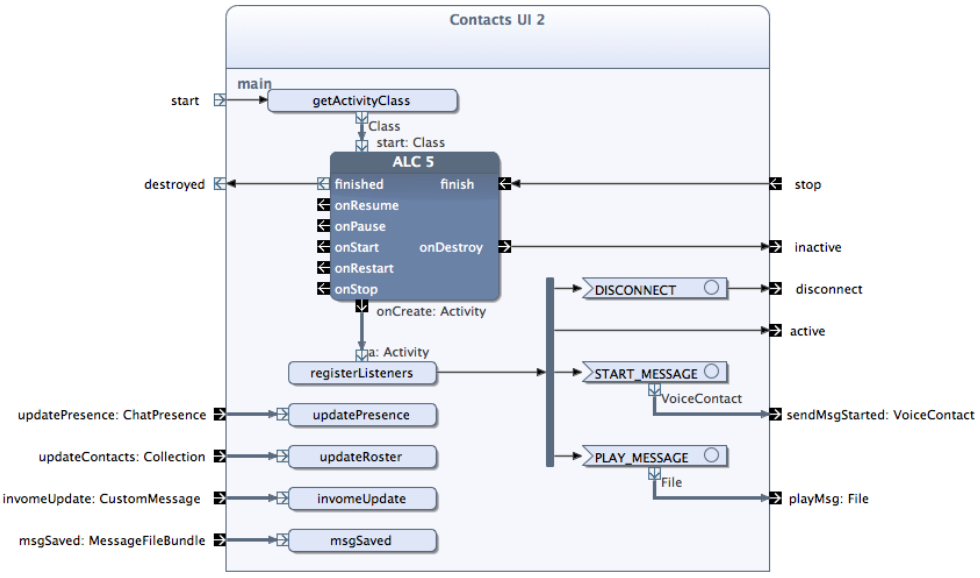
A new Activity Life Cycle block has been created, *ALC 5*, with a flat ESM. Previous ALC blocks have sought to recreate the state of an activity as ESM states, but Android does not guarantee that the *onStop()* and *onDestroy()* methods are called when the activity is killed [28], so a flat ESM prevents the block from ending up in a state inconsistent with the actual activity state. The downside of this approach is that the Arctis analyzer gives six warnings of the form "Flow stopped at <pin>". Every one of these situations has been examined manually and found not to cause any problems in the program flow.

Once the block is started, the *Class* of our *ContactListActivity* is sent to *ALC 5* for initialization. The actual user interface (seen in Figure 4.4) is specified in XML files, the main view with the list in *contacts_main.xml* and the layout for a single contact in *single_contact.xml*. Once the block has been created, *registerListeners()* is called, which sets up the dialog that is displayed when a contact is selected. When the listeners are in place, *active* fires and the block is ready to receive updates, process them, and update the screen.

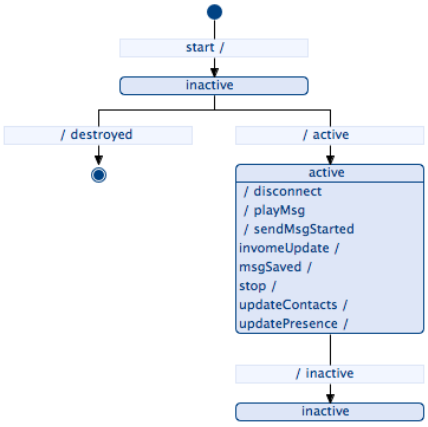
When a new message has been saved, the *msgSaved*-pin fires and the sending contact will get a "mail"-icon in the contact list, denoting the number of stored messages. All UI input events are represented as Arctis signals and will fire their respective output-pin when triggered, sending signals to the environment.

4.4 To Service

By gathering all service communication functionality in a single block, not only do we make the communication easy to visualize, we also hide the specific communication implementation from the main block; we could at any time create a

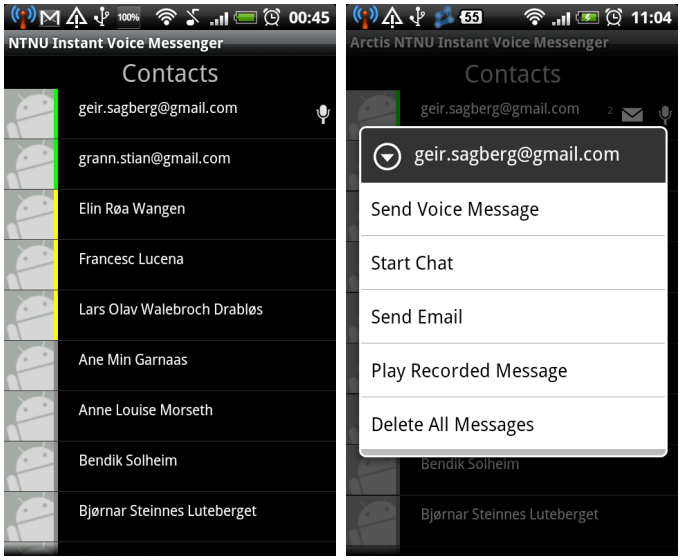


(a) Internal behaviour



(b) External State Machine

Figure 4.3: *Contacts UI 2* behaviour and ESM



(a) A list of contacts with presence and voice status

(b) Displaying options for a selected contact

Figure 4.4: *Contacts UI 2* user interface

new pair of communication blocks based on e.g. a *Messenger* interface, without having to change any other parts of the model.

The internal behaviour of the block can be seen in Figure 4.5; the ESM is not shown, as the block has only a single state, and all transitions are trivial. Once the block is started with the *start*-pin, the service is started with the *startService()* operation. A broadcast receiver is initialized with a list of the action strings it should listen for, which we have specified in a separate interface, *Constants. To Service* and *To GUI* both implement this interface, to ensure correct spelling of the action strings.

After receiver initialization, the service is polled to find out whether it is connected to the XMPP server already. As the service may not have started yet, we poll it using *getContext().sendStickyBroadcast(intent)*. A sticky broadcast is stored by the system until a matching receiver appears, so we are ensured to get a response as soon as the service is up and running.

All outgoing communication is shown on the left side of the model, with pins *connect, playMsg, disconnect, setStatus* and *stopPlaying*. Each pin is connected to an operation, which broadcasts an intent with the associated action, bundling

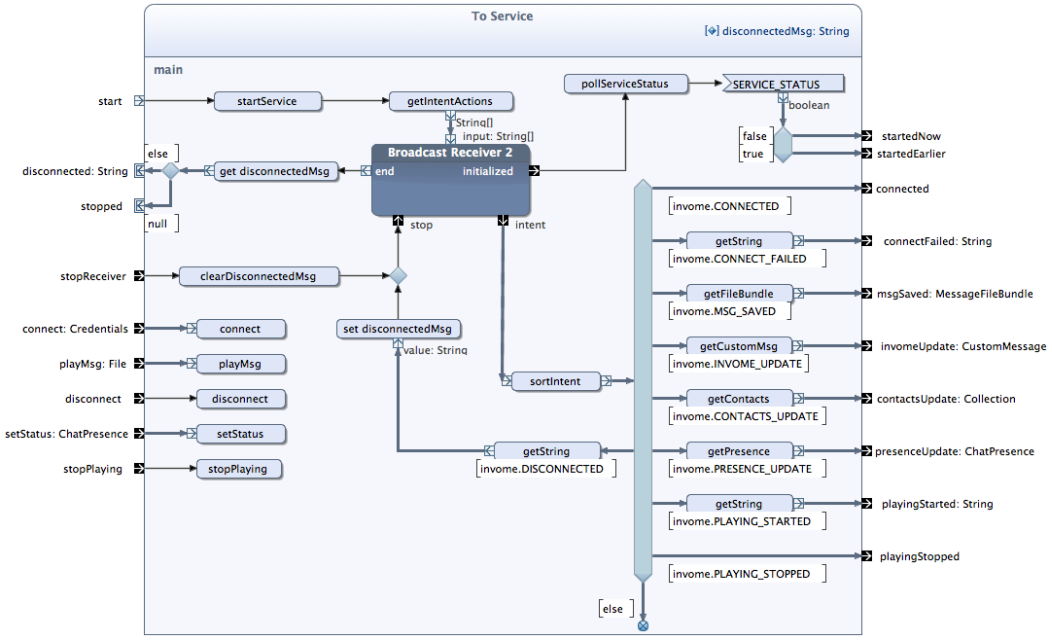


Figure 4.5: Internal behaviour of *To Service*

data if necessary.

Similarly, incoming intents are fired through the broadcast receiver's *intent*-pin, and sorted on the right side of the model. Only a received *DISCONNECTED* intent is treated differently, causing the receiver to be stopped and unregistered, and firing the *disconnected*-pin with an error message if present.

4.5 General Optimizations

In accordance with the techniques for improved responsiveness presented in Chapter 2.1, and the best practices in Chapter 2.2.3, the following measures have been taken:

- All buttons give visual feedback
- Contact list is refreshed continually as new contact updates arrive
- A progress dialog is displayed during the only extended operation, the login process

Table 4.1: State space comparison of old and new blocks

Old blocks	States	Transitions	New blocks	States	Transitions
InVoMe	33	586	InVoMe Service	13	213
			NTNU Instant Voice Messenger	2	20
Contacts UI	19	83	Contacts UI 2	7	44
XMPP Client 4	11	76	XMPP Client 5	10	27

- No network calls or blocking operations have been placed in the main UI thread
- Every dialog and activity has been made cancelable
- Runtime errors are caught and displayed as well-formed alert dialogs
- When the application is hidden, only notifications and toasts are shown

4.5.1 State Space and Code Measurements

As a result of the redesigned blocks, the total state space has shrunk considerably. The state space and transition differences for the most important blocks can be seen in Table 4.1. For example: *InVoMe*, the main system block of the previous version, had 33 states and 586 transitions, while the two system blocks of the new version have a total of 15 states and 233 transitions.

These improvements, along with marking most blocks as single-session blocks, translate into a considerably smaller code amount: The generated state machine code for the old version has a total of 4133 lines, while the new version's generated code consists of 1173 lines for the service and 1301 lines for the foreground, totaling 2474 lines. This again translates into a smaller application that uses less system memory.

Chapter 5

Background Service

This chapter will describe the building blocks used in the service part of our application, starting with an overview of the main model and the general program flow, and then covering each of the building blocks in turn, highlighting improvements and new features versus the old InVoMe application. Finally, known issues will be covered and solutions will be discussed.

5.1 Main Service Model Overview

As can be seen in Figure 5.1, the service model is a bit more complex than the user interface model. This was necessary due to all the functionality contained in the service, so the model does not follow the same top-down program flow as the foreground model; we instead decided to place the service-activity communication functionality to the left in the model and notifications to the right, with all the rest of the functionality in the middle. This was found to be a good placement for minimizing the number of crossing event flows, while still having the program flow start in the top left as in the foreground model.

Once the service has been started, the sticky broadcast from the foreground is received and answered, and *To GUI* will fire *connect*, sending credentials to the *XMPP* block for logging in. A successful connection fires the *connected* pin, which both notifies the GUI and sets up the *Receiver* with a listening port. As soon as the *Receiver* is initialized and listening for incoming RTP connections, *Service Controller* is used to move the service to foreground mode, with a *Notification-Bundle* containing the icon and text to be shown in the status bar.

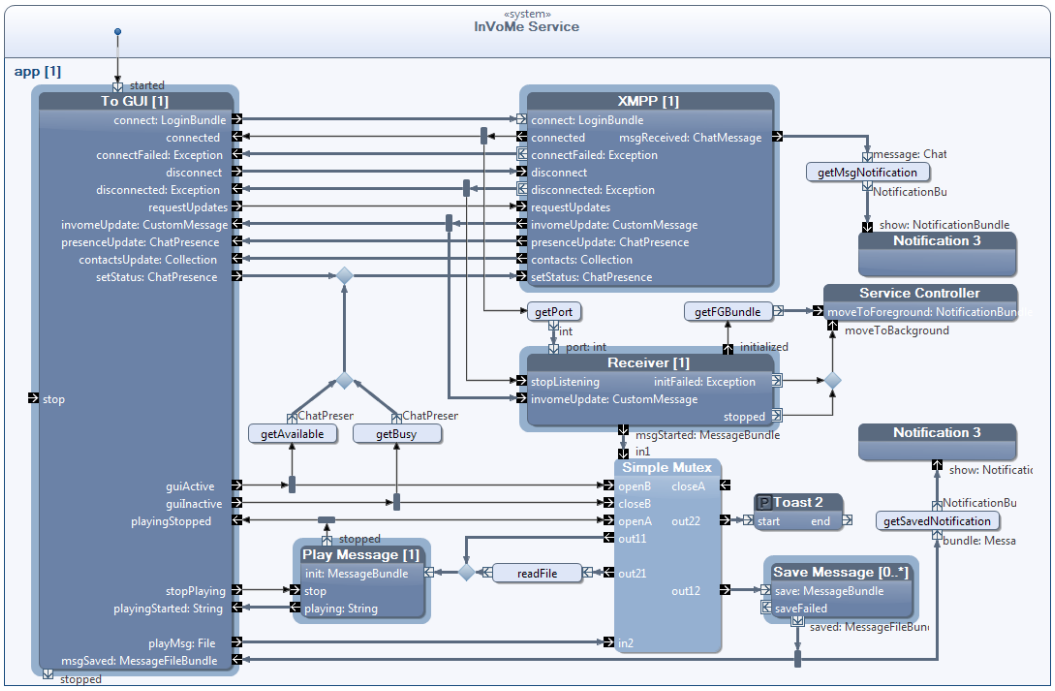


Figure 5.1: Internal behaviour of the background service part of *NTNU Instant Voice Messenger*

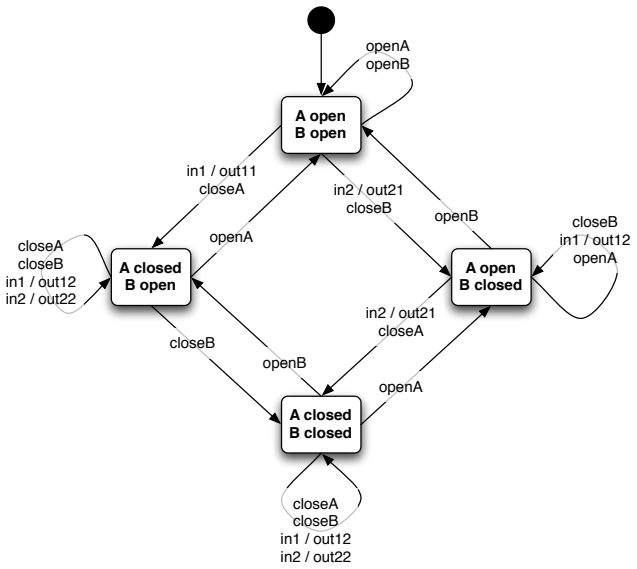


Figure 5.2: State diagram of *Simple Mutex*

All kinds of updates from *XMPP* are sent directly to GUI via *To GUI*. Contrary to InVoMe, this application reacts to incoming chat messages, fired from the *msgReceived*-pin of *XMPP*. When a chat message is received, a notification is displayed with the sender name and a message preview, with an option for opening the chat in a suitable application, e.g. Google Talk.

For controlling whether incoming voice messages should be played or stored, we created a shallow block, *Simple Mutex*. Figure 5.2 shows how input signals are routed. The block has two locks A and B, two input pins *in1* and *in2*, and two output pins per input pin; flows entering *in1* can exit through either *out11* or *out12*, while flows entering *in2* can exit through either *out21* or *out22*. Input *in1* is used for received voice messages, while *in2* is used when the user has selected to play a stored message. A is open if and only if the audio device is ready, and B is open if and only if the user interface is active. Incoming messages (*in1)* can only be played when both the audio device is ready and the user interface is showing. Saved messages (*in2*) are played as long as the audio device is ready.

Incoming messages that are not played at once are instead passed via *out12* to *Save Message*. *Save Message* is marked as a multi-session block, as can be seen in the title *(Save Message [0..*])*. This means that several instances of the block can be started and run in parallel. In this way, every message that is received is

either played back instantly or stored safely to SD card for later playback.

5.2 XMPP Blocks

To keep the main model as simple as possible, *XMPP Client 5* and *XMPP Gateway* were wrapped in a single block, *XMPP*, seen in Figure 5.3. Like the *Voice Support* block of InVoMe, *XMPP Gateway* translates between XMPP-specific and generic objects, prompts every contact for voice compatibility when a roster update arrives, and answers *IP_REQUEST* custom messages.

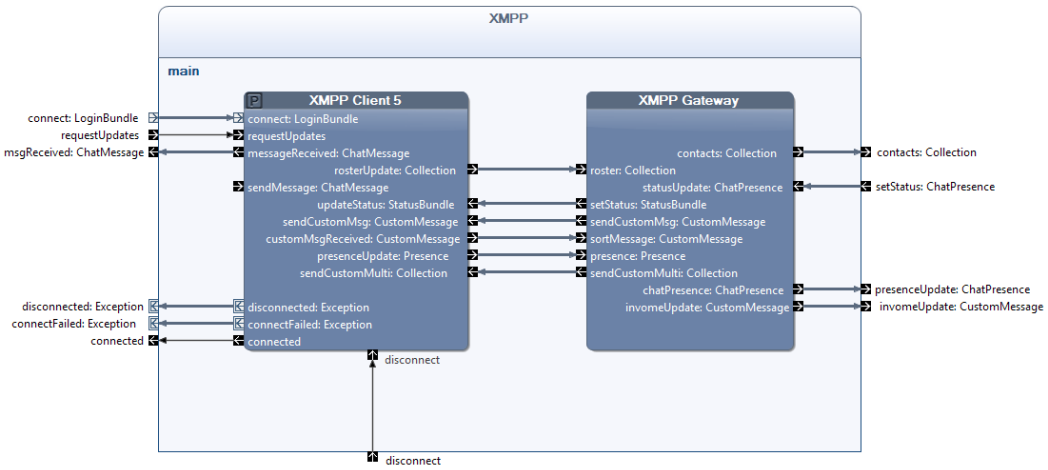
5.2.1 *XMPP Client 5*

The internal behaviour of *XMPP Client 5* can be seen in Figure 5.4; the ESM is not shown as it is roughly equivalent to the *XMPP* ESM shown in Figure 5.3b. The model is based on *XMPP Client 4* shown in [4, Ch 4], but has been redesigned for a cleaner state space and better understandability. The previous version had several blocking operations where synchronous network calls were performed in the main thread, causing the Arctis runtime scheduler to freeze momentarily. In this version, every operation containing network calls uses *Runnables* to run the critical code in a background thread.

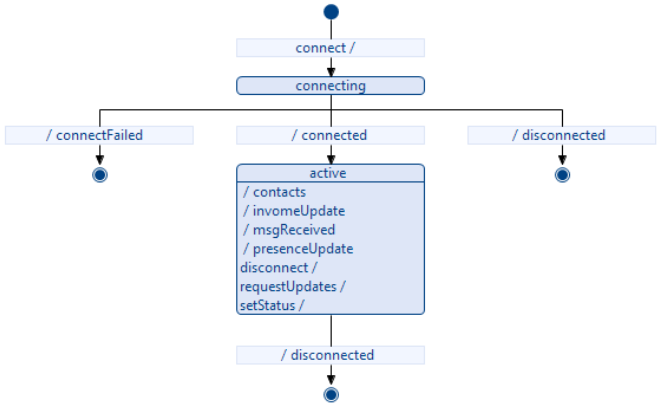
The block is started by sending a *LoginBundle* with credentials to the *connect*-pin. The operation *connectToServer()* creates a new thread for registering update listeners and connecting to the XMPP server, communicating the result back to Arctis by the signals *LOGIN_SUCCESSFUL* or *LOGIN_FAILED*. A successful login fires the *connected*-pin, and the block is ready to receive XMPP updates. If at any time the connection to the server is lost, the *CONNECTION_CLOSED* signal is sent, with an exception detailing the error.

Because of how the block is programmed, either *LOGIN_SUCCESSFUL* or *LOGIN_FAILED* can be received, not both. However, the Arctis analyzer does not know this, so a "Flow stopped at 'connectFailed"'-warning is given, highlighting a state when *LOGIN_FAILED* is received after the block is already in the *connected* state. As we know this will never happen, this warning can safely be ignored.

Outgoing messages are shown on the left side of the model, with pins *updateStatus, sendMessage, sendCustomMsg, sendCustomMulti* and *requestUpdates*. Every pin has its associated operation with blocking operations wrapped in *Runnables* and



(a) Internal behaviour



(b) External State Machine

Figure 5.3: *XMPP* behaviour and ESM

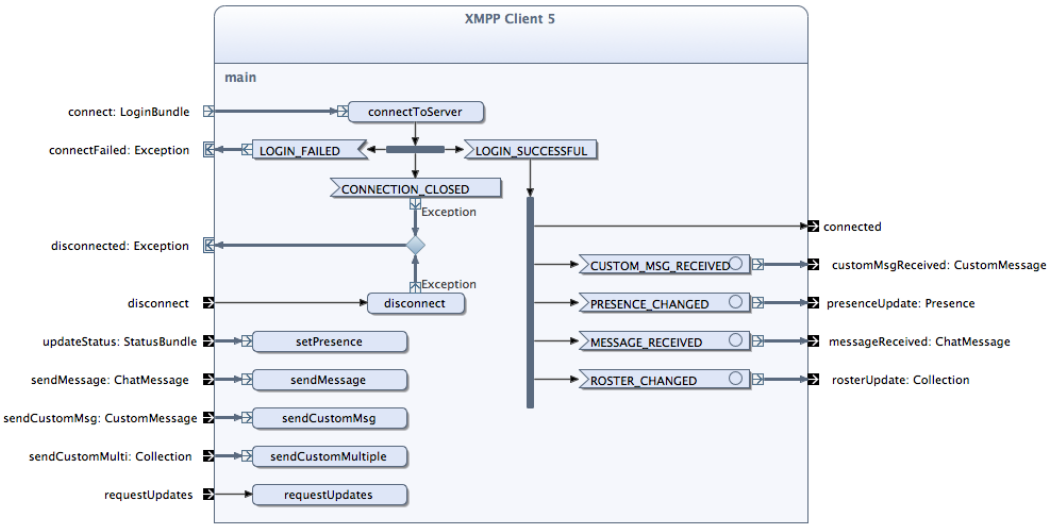


Figure 5.4: Internal behaviour of *XMPP Client 5*

output sent as the signals on the right side of the model; for example, when *requestUpdates* fires, a new thread is started, the contact roster is retrieved and sent as a *ROSTER_CHANGED*-signal, then the contact presences are retrieved one by one and sent as *PRESENCE_CHANGED*-signals.

5.2.2 XMPP Gateway

The internal behaviour of the stateless block *XMPP Gateway* can be seen in Figure 5.5. Not much has changed from *Voice Support* in InVoMe; we have added a translation from the generic *ChatPresence* class to the XMPP-specific *StatusBundle*, and removed the pins for handling multiple presence updates at once, as this was no longer necessary and increased the model complexity of both this block and *XMPP Client*. The rest of the functionality is as described in [4, Ch 6.3].

5.3 Message Blocks

In InVoMe, incoming voice messages were handled by a single block, *Audio RTP Receive 2*. This block used *RTP Receiver 2* and *Play Audio 2* to play every incoming message, and did not support more than one received message at once.

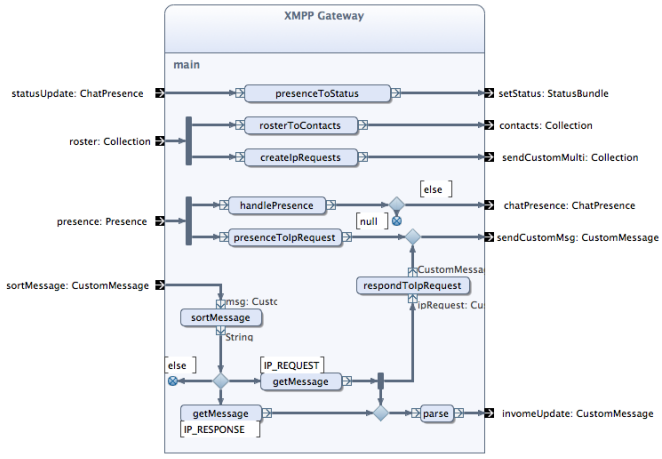


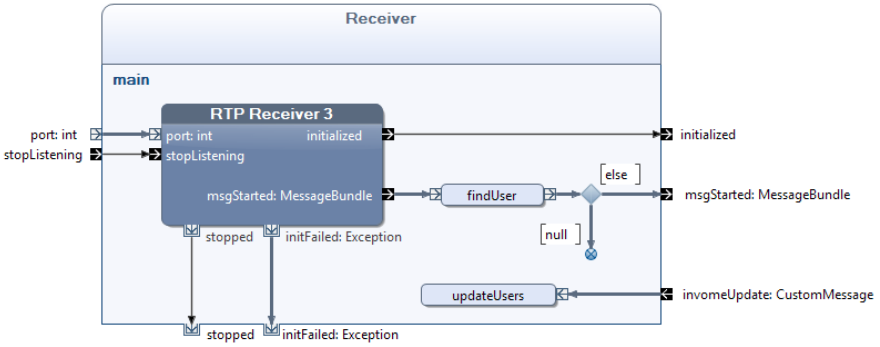
Figure 5.5: Internal behaviour of *XMPP Gateway*

In order to allow incoming messages to be either played or stored, we had to split the functionality in three: one *Receiver* block for listening for RTP connections and creating new *InputStream* objects for every new connection, one *Play Message* block for playing messages, and one *Save Message* block for storing messages.

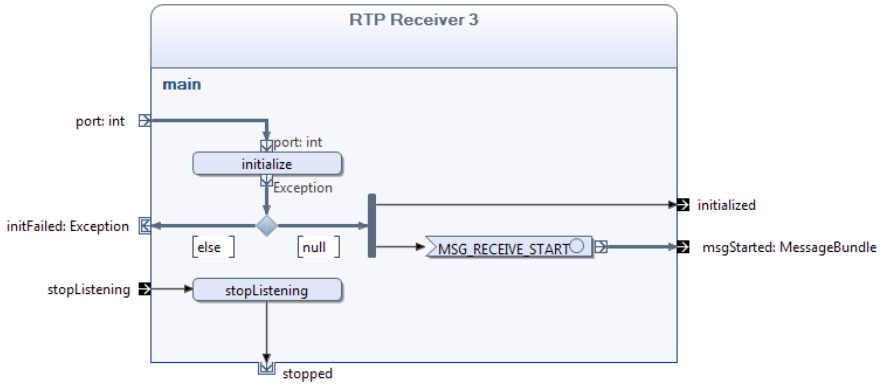
5.3.1 Receiver

The internal behaviour and ESM of *Receiver* and *RTP Receiver 3* can be seen in Figure 5.6. The block is initialized by passing the RTP listening port. In the previous version, an *InputStream* was passed alongside the port number, and that single stream was used for every received message. As every RTP packet was printed to the same stream indiscriminately, two incoming voice messages at the same time would result in garbled audio intermixed from the two messages. To fix this, *RTP Receiver 3* was created, which creates a new *InputStream* whenever it receives an RTP packet from a new IP address, and sends it together with the IP address as a *MessageBundle* through the *msgStarted*-pin.

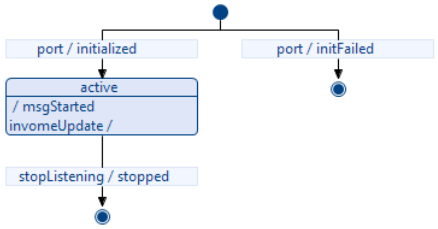
When a message arrives and needs to be stored, our first thought was to create a filename by using the sending IP address and a timestamp. However, we need to know which user the message is from, and IP addresses may change without notice. We therefore added a mapping between current IP addresses and usernames, so the sender's username could be used both in notifications and as part of the filename. As can be seen in Figure 5.1, whenever an *invomeUpdate* arrives from



(a) Internal behaviour of *Receiver*



(b) Internal behaviour of *RTP Receiver 3*



(c) External State Machine of *Receiver*

Figure 5.6: *Receiver* and *RTP Receiver 3* behaviour and ESM

XMPP, it is forked and sent both to *To GUI* and *Receiver*.

Another new feature is the timeout detection and end-of-stream detection added to *RTP Receiver 3*. When no new packet has arrived for a given IP address after a timeout delay of half a second, the input stream is closed. The end-of-stream detection works when this block is used conjunction with *RTP Sender 3* in *Audio RTP Send 2*; *RTP Sender 3* sends a special signal as part of the RTP stream, which *RTP Receiver 3* recognizes and promptly closes the input stream for the given sender.

5.3.2 Play Message

Play Message was created as a simple wrapper around *Play Audio 3*, in order to keep the main model more organized. Figure 5.7 shows the internal behaviour of both blocks and their ESM, which is the same for both blocks.

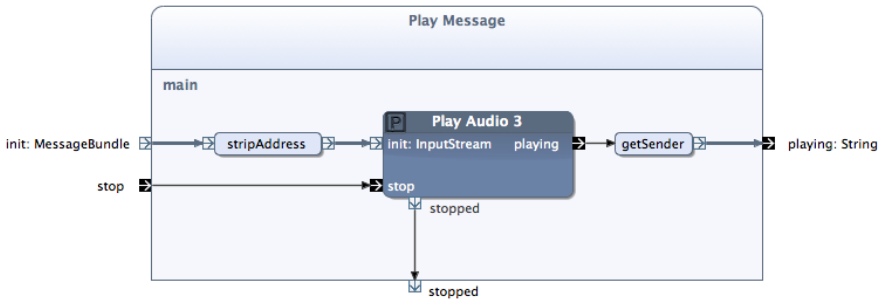
While *Play Audio 3* was created for generic use, *Play Message* is started with an application-specific *MessageBundle*, containing the input stream to read the message from, and the IP address or username of the sender. The input stream is extracted from the bundle and fed to *Play Audio 3*, while the sender is stored and sent as a string via *playing*, so the sender's username can be displayed in a "Playing..." dialog.

Play Audio 3 is based on *Play Audio 2* used in InVoMe, but while the previous version relied on external signals to stop the playback, *Play Audio 3* will stop automatically if end of stream is reached. This simplifies the program flow and is more intuitive.

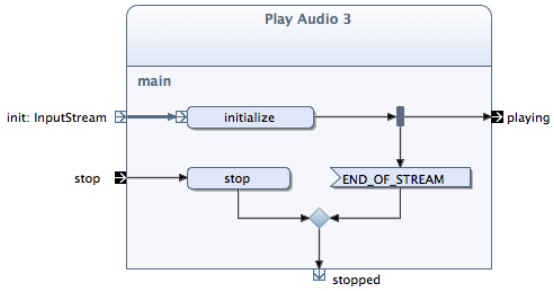
5.3.3 Save Message

Like *Play Message*, *Save Message* is an application-specific wrapper, using the generic block *Write File* to store messages to SD card. Figure 5.8 shows both models and their shared ESM.

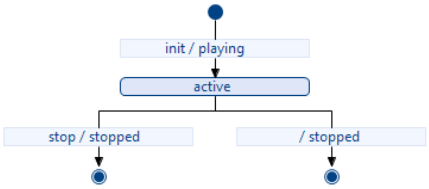
When a message is to be saved, the operation *createFilename()* uses the sender's username and a timestamp to create a unique filename, which is passed to *Write File* together with the message input stream. Once *Write File* encounters the end of the stream, it sends a *WRITE_TO_FILE_FINISHED* signal, and *Save Message* sends a *MessageFileBundle* containing the sender's username and the resulting file.



(a) Internal behaviour of *Play Message*

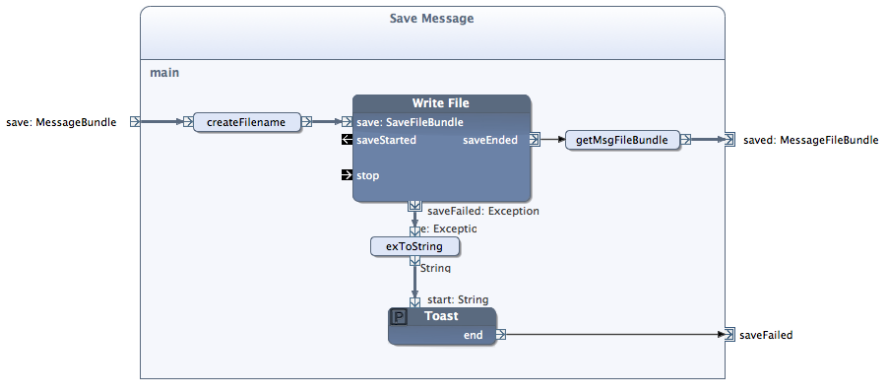


(b) Internal behaviour of *Play Audio 3*

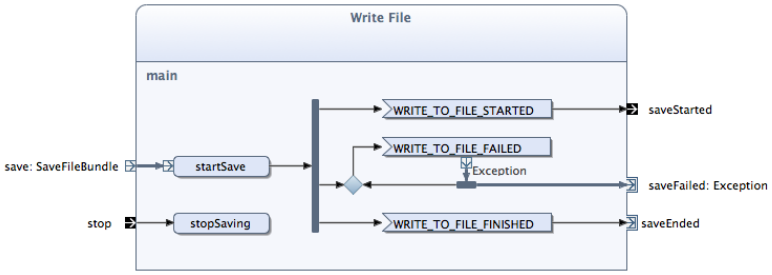


(c) External State Machine of *Play Message*

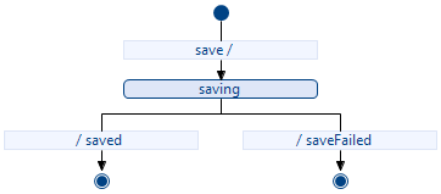
Figure 5.7: *Play Message* and *Play Audio 3* behaviour and ESM



(a) Internal behaviour of *Save Message*



(b) Internal behaviour of *Write File*



(c) External State Machine of *Save Message*

Figure 5.8: *Save Message* and *Write File* behaviour and ESM

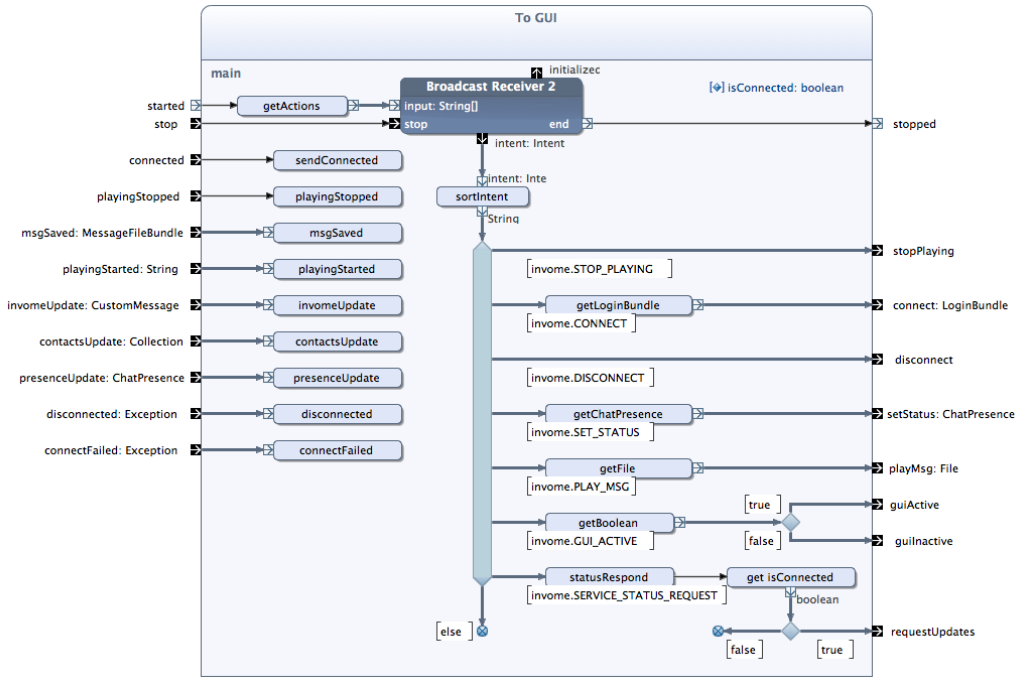


Figure 5.9: Internal behaviour of *To GUI*

If the message failed to save for any reason, the error message is displayed as a toast, and the block terminates via the *saveFailed*-pin.

5.4 To GUI

The equivalent of *To Service* in the foreground model, this block handles all the communication with the foreground activity. Figure 5.9 shows the internal behaviour of the block. After the block is started, a *Broadcast Receiver 2* is initialized with the list of actions specified in the *Constants* interface.

Like its counterpart, *To GUI* consists of incoming intents handled on the right side of the model, and outgoing intents on the left side. Most actions are self-explanatory, the only outgoing objects that are not sent directly are the *Exceptions* of the pins *disconnected* and *connectFailed*. Instead, a string is sent ("Disconnected from server" or "Could not connect to server", respectively) with the exception message appended if it is not empty.

Once the receiver has been initialized, there is probably a sticky broadcast from the foreground activity waiting, with the action *SERVICE_STATUS_REQUEST*. The variable *isConnected* is set to *true* as part of the *sendConnected()* operation, and to *false* in *disconnected()* and *connectFailed()*. In this way, we only have to respond to the service status poll by sending back the value of *isConnected*. If the service is already connected, *requestUpdates* fires, requesting an updated contact list and presences from *XMPP*.

5.5 Areas of Improvement

While we have aspired to make our application as complete and stable as possible, there are still points that might be improved; we will present them here with suggested solutions.

5.5.1 Security

As every Android process keeps a private memory space, we do not need to worry about other applications snooping on our memory. However, for communicating between foreground and background, we broadcast intents system-wide. This could be dangerous, as any other application could register itself to listen to these intents, and snap up a user's Google credentials. However, this is avoided by imprinting the broadcasted intents with a signature permission, so only our own processes can receive them. But if a phone is rooted, no guarantees are made that the broadcasted intents are safe from malicious applications.

Another aspect of the Android system is the storage; as many phones have limited internal flash storage, we use the SD card for storing messages. However, data stored on the SD card can be read by any other application, so in theory a malicious application could read the stored voice messages and upload them to the Internet. To alleviate this, we would have to store the messages encrypted; however, this issue was not considered vital enough for us to spend the time to implement message encryption.

In our application, the user has to enter their Google username and password directly into our application. This means that the user must trust us enough not to store their credentials and use them for malicious purposes. It was done this way because the XMPP library we used, *smack*, only supports username/password

authentication; however, the XMPP protocol supports authentication by token [29], so if a library was found that implemented this, or *smack* was updated to support it, our application could use Google's OAuth API [30] to authenticate users. In this way, they would not have to trust us with their credentials.

5.5.2 RTP

Currently, the application listens for any incoming packets on the RTP port, and creates a new *InputStream* whenever a packet from a new IP address is received. While a mapping between known contacts and their IP addresses keeps unauthorized packets from being played as audio or stored as messages, the RTP block could be abused by sending streams from several IP addresses at once, causing numerous input streams to be created, and possibly using too much computational resources for the application to correctly handle the genuine connections. This scenario was considered unlikely enough not to cause a problem, but could be prevented by modifying the RTP blocks to keep a safe list of IP addresses, and discard any packets from unknown sources.

Chapter 6

Conclusion

In this report, we have presented the development of *NTNU Instant Voice Messenger*, a responsive and stable communication application for Android. The application supports communication with other Google Talk contacts using voice messages, text chat or email. The application is a major improvement on the previous version, InVoMe, and has been considered good enough to be the first Arctis-developed application released on the Android Market.

The application has been developed with a specific focus on providing a responsive, seamless and intuitive user experience, as well as fit well into the Android application environment. We have presented both theory and techniques for improving the responsiveness of an application. General user interface design concepts have been presented, and specific measures for both Android and Arctis have been researched and explained.

The Android application structure has been researched to gain an understanding of how an application created in Arctis can run as an Android service. The various service implementations and communication models have been examined, prototype applications have been developed and explained, and we have presented a general methodology for merging two Arctis system models - a background service and a foreground user interface - into a single Android application.

Several reusable building blocks have been developed, including blocks for writing to and reading from the file system, blocks for showing status bar notifications, and blocks for displaying dismissable dialogs. Existing building blocks have been improved with cleaner layouts, smaller state spaces and non-blocking operations.

Bibliography

[1] Nokia, "Symbian at Nokia." http://symbian.nokia.com/. Accessed 11.06.2011.

[2] Open Handset Alliance, "What is Android." http://developer.android.com/guide/basics/what-is-android.html. Accessed 13.12.2010.

[3] Apple Inc., "iOS Dev Center." http://developer.apple.com/devcenter/ios/index.action. Accessed 11.06.2011.

[4] G. Sagberg, "Voice-Based Group Communication System on Android," project thesis, Norwegian University of Science and Technology, 2010.

[5] F. A. Kraemer, "Arctis: Composing M2M Applications." http://www.thinkarctis.com/. Accessed 13.12.2010.

[6] Google, "NTNU Instant Voice Messenger - Android Market." https://market.android.com/details?id=no.ntnu.item.arctis.android. Accessed 11.06.2011.

[7] Vimeo, LLC., "NTNU Instant Voice Messenger on Vimeo." http://vimeo.com/24932676. Accessed 11.06.2011.

[8] GitHub Inc., "arctis/students." https://github.com/arctis/students. Accessed 16.06.2011.

[9] H. Schulzrinne et al, "RTP: A Transport Protocol for Real-Time Applications." http://www.ietf.org/rfc/rfc3550, July 2003. Accessed 10.12.2010.

[10] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core." RFC 3920 (Proposed Standard), Oct. 2004.

[11] F. A. Kraemer, "Arctis and Ramses: Tool Suites for Rapid Service Engineering," in *Proceedings of NIK 2007 (Norsk informatikkonferanse), Oslo, Norway*, Tapir Akademisk Forlag, November 2007.

[12] K.-A. Martinsen, "Encapsulation of Android User Interfaces in Arctis," project thesis, Norwegian University of Science and Technology, 2009.

[13] J. Nielsen, *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[14] Open Handset Alliance, "Designing for Responsiveness." `http://developer.android.com/guide/practices/design/responsiveness.html`. Acccessed 04.06.2011.

[15] Open Handset Alliance, "Thread." `http://developer.android.com/reference/java/lang/Thread.html`. Accessed 11.06.2011.

[16] Open Handset Alliance, "AsyncTask." `http://developer.android.com/reference/android/os/AsyncTask.html`. Accessed 06.06.2011.

[17] Open Handset Alliance, "ProgressDialog." `http://developer.android.com/reference/android/app/ProgressDialog.html`. Accessed 11.06.2011.

[18] Open Handset Alliance, "Application Fundamentals." `http://developer.android.com/guide/topics/fundamentals.html`. Accessed 06.06.2011.

[19] Open Handset Alliance, "The AndroidManifest.xml file." `http://developer.android.com/guide/topics/manifest/manifest-intro.html`. Acccessed 13.06.2011.

[20] Open Handset Alliance, "Parcel." `http://developer.android.com/reference/android/os/Parcel.html`. Accessed 18.06.2011.

[21] Open Handset Alliance, "Android Interface Definition Language (AIDL)." `http://developer.android.com/guide/developing/tools/aidl.html`. Acccessed 15.06.2011.

[22] Open Handset Alliance, "Designing for Seamlessness." `http://developer.android.com/guide/practices/design/seamlessness.html`. Acccessed 11.06.2011.

[23] Open Handset Alliance, "Tasks and Back Stack." http://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html. Acccessed 13.06.2011.

[24] Open Handset Alliance, "Multitasking the Android Way." http://developer.android.com/resources/articles/multitasking-android-way.html. Acccessed 13.06.2011.

[25] SourceForge.net Open Source Community, "RTP Java Library." http://sourceforge.net/projects/jlibrtp/. Accessed 13.12.2010.

[26] Open Handset Alliance, "DatagramPacket." http://developer.android.com/reference/java/net/DatagramPacket.html. Accessed 14.12.2010.

[27] Open Handset Alliance, "Service." http://developer.android.com/reference/android/app/Service.html. Accessed 18.06.2011.

[28] Open Handset Alliance, "Activity." http://developer.android.com/reference/android/app/Activity.html. Accessed 17.06.2011.

[29] XMPP Standards Foundation, "XEP-0235: OAuth Over XMPP." http://xmpp.org/extensions/xep-0235.html. Accessed 20.06.2011.

[30] Google Inc., "Authentication and Authorization for Google APIs." http://code.google.com/apis/accounts/docs/GettingStarted.html. Accessed 20.06.2011.