Disfer

Sink - Sensor Connectivity and Sensor Android Application

Protocol implementation: Charilaos Stais (stais AT aueb.gr)

Android development: Dimitri Balerinas (dimi.balerinas AT gmail.com)

Supervised by: George Xylomenos (xgeorge AT aueb.gr)
The goal of this project is to offer a first implementation of the protocol introduced in the paper, Sink Controlled Reliable Transport for Disaster Recovery by Charilaos Stais, George Xylomenos and Giannis F. Marias, on Android environment. The implementation is eventually a user friendly Android application which is going to be used to support the Disfer project in order to run tests and experiments.

The main logic of the implementation will be analyzed in this report as well as a description of the Android application and its use.
Sink - Sensor Connectivity

Why Java

Both base protocol implementation and Android Application were developed in Java. The language provided all necessary resources for a most optimal version of the protocol that later was imported into an Android Development environment and was thoroughly and repeatedly tested and compiled for deferent Android API levels and later on Virtual and Physical devices through an installable application form.

How does the protocol work

The protocol which was imported [1] implements a communication between sensors and a sink during a procedure of five stages: connection establishment, sensor information exchange, data exchange, idle and connection release. Quoting from the paper itself, the application of the five stages is described bellow:

3.1 Connection establishment

![Diagram of connection establishment](image)

Figure 1: Connection establishment stage.

When the sink begins operation, it listens to a well-known UDP port for connection requests from sensors. The sensors wait until the sink becomes reachable before connecting to it. In the protocol prototype introduced, the sink periodically sends a probe message, MSG_HELLO, to the well-known IP address and UDP port of the sink, until it receives a response; the probe interval is configurable. If the routing protocol supports it, the sink can ask the routing engine to be notified when the sink becomes reachable.
The MSG_HELLO message includes the sensor identifier, unique in the sensor network, and its type; in the protocol prototype, only event and continuous sensor types exist. When the sink receives such a message, it responds with an MSG_HELLO_ACK message which indicates a separate UDP port where the sensor should send the following control messages. The sink notes the time when it sent the message, so as to later measure the round-trip time (RTT) to that sensor. The use of a separate control port per sensor allows the sink to dedicate one thread to receiving new connection requests, and a separate thread for each connection to a specific sensor, thus avoiding the need to multiplex messages from multiple sensors over a single control connection.

The sink notes the time when it sent the message, so as to later measure the round-trip time (RTT) to that sensor. The use of a separate control port per sensor allows the sink to dedicate one thread to receiving new connection requests, and a separate thread for each connection to a specific sensor, thus avoiding the need to multiplex messages from multiple sensors over a single control connection.

The connection establishment messages are depicted in Figure 1.

### 3.2 Sensor information and idle

After receiving a response from the sink, the sensor prepares an MSG_INFO message which includes the delay between receiving the message from the sink and sending the response, the data rate requested by the sensor, whether the sensor is ready to send data at this point in time or not, the data packet size and the total size of the data to send.

When the sink receives the MSG_INFO message it calculates the time elapsed since sending the MSG_HELLO_ACK, subtracting the delay in the message to get the RTT to the sensor. At this point, the connection has been established and the sink is aware of the sensor’s bandwidth requirements.

If the sensor has indicated that it is not ready to send data at this point, the sink moves to an idle state, waiting until the sensor sends a MSG_CTRL_DATA with no parameters. Then, the sink starts the data exchange by sending an MSG_CTRL_START message to the sensor, indicating the data rate to use and a UDP data port to use for the transmission.

If the sensor indicated in the MSG_INFO message that it was ready to send data, the MSG_CTRL_START message is sent immediately as a response. The initial rate allocated to the sink is set as explained in Section 4.

The sensor information messages are shown Figure 2.
3.3. Data exchange

After the sink sends the MSG_CTRL_START, the actual data transfer begins, using the UDP data port assigned for the transfer; control messages, such as NACKs and rate updates, are exchanged out of band over the control channel, without being rate controlled. This allows control messages to be sent without waiting behind a, possibly long, queue of data messages. The sensor breaks down its transmission into packets with the size indicated in the MSG_INFO message, until all the data indicated in the MSG_INFO message is exhausted. Data packets only have a single header field, a segment number used to sequentially number all data packets. When a missing packet is detected, the sink sends a MSG_NACK to the sensor over the control channel. However, the sensor does not immediately retransmit lost messages. After the transmission is complete, all missing messages are retransmitted, generating further NACKs from the sink, if needed. This procedure is repeated in rounds, until all messages are received [2]. Once recovery is complete, the sink sends a MSG_CTRL_DONE message to indicate a successfully completed data transfer. Both endpoints then move to an idle state, until the sensor generates a new MSG_CTRL_DATA message. If the need arises, the sink will send a MSG_CTRL_RATE message to the sensor indicating its new rate allocation, as explained in Section 4.

The data messages are shown in Figure 3.

Note that round-based recovery allows the sink to use any packets received without waiting for retransmissions. The sink may even stop the recovery process by sending the MSG_CTRL_DONE message. For example, when an image is transmitted using redundancy coding, the sink may stop the recovery process when enough packets have been received to adequately reconstruct the image. This allows the application to fine tune the reliability of the protocol.
3.4 Connection control and release

Since the path between the sink and the sensor may become disconnected due to the sink’s mobility, the connection may fail between data transfers, without either side noticing. For this reason, the sink periodically sends an MSG_CTRL_ALIVE message to the sensor, which is acknowledged by an MSG_CTRL_ACK message from the sensor that includes the delay incurred between receiving the MSG_CTRL_ALIVE and responding with the MSG_CTRL_ACK. In addition to confirming that the connection is still alive, this procedure allows the sink to periodically measure the RTT of the connection.

If the sensor or the sink wishes to complete the connection, they can send a MSG_CTRL_BYE message, which does not need to be acknowledged, as after either side drops the connection, the other one will eventually timeout: the sink times out if no responses are received to its MSG_CTRL_ALIVE messages, while the sensor times out if no MSG_CTRL_ALIVE messages arrive.

The MSG_CTRL_BYE message simply allows the other end to release the resources dedicated to the connection without waiting for a timeout.

The connection control messages are shown in Figure 4.

Figure 4: Connection control and release stage.
One of the most important features of the protocol that was imported in this project [1] is the congestion management implementation.

The congestion management mechanism of the protocol is agile and purely sink-driven. The congestion is focused around the sink, and total available bandwidth is known as it depends on the technology used by the sink and sensors for data exchange. At first, a fixed part of this bandwidth is being reserved, e.g. 30%, for the control message exchanges which are not rate-controlled. Then, the sink splits the remainder to event and continuous sensors using a ratio determined by the application, e.g. 10%-90%, depending on the number and type of sensors present at the disaster site.

The congestion management algorithm periodically evaluates the state of individual connections and the system as a total. The sink monitors the RTT of each connection using the MSG_HELLO_ACK, MSG_INFO, MSG_CTRL_ALIVE and MSG_CTRL_ACK messages; the processing delay at the sensor is always subtracted to get an accurate RTT estimate. The congestion management algorithm maintains the last few RTT samples and their moving average. Whenever the algorithm runs, it first checks whether the average for each sensor has increased compared to the previous value by more than a configurable threshold. If this occurs four times in a row, then the corresponding connection is congested, otherwise it is not. If the connection is congested, the sink instructs the sensor to reduce its rate by 20%, via a MSG_CTRL_RATE message. After each individual sensor is checked, if a new sensor has been connected or an existing one has been disconnected, the entire system is checked to see whether global adjustments need to be made.

This takes place separately for each sensor class. First, the total rates requested (not assigned) by the sensors of the class are being calculated. If these are below the available bandwidth, they will all get what they asked for. New sensors will get their requested rate in the MSG_CTRL_START message which directs them to start sending data. Sensors that were previously rate limited, will increase their rate by 20%, while other sensors will get their requested rate; in both cases, the change is announced via a MSG_CTRL_RATE message. If, on the other hand, the requested bandwidth is higher than the available one, the available rate is shared equally among all sensors of that class. The sensors are notified as above, i.e. either via a MSG_CTRL_START or via a MSG_CTRL_RATE message.

The congestion management algorithm is very simple, as congestion is expected to be concentrated around the sink. Since the sink is constantly on the move, it is very unlikely that a distributed congestion control management will have time to converge. While TCP uses an Additive Increase - Multiplicative Decrease (AIMD) algorithm, the scheme uses fixed and symmetric steps. This is because TCP sources constantly probe the network for capacity, hence entering deep into the congested region before having to abruptly backoff. In the scheme sensors are conservatively rate controlled, hence congestion is expected to appear slowly, therefore there is no need for dramatic rate reductions.
Android Application

In order for the Android Application implementation to occur, the main protocol infrastructure was imported, edited and enriched in order to stay functional and apply to the needs of a properly working User Interface that included an environment designed to be optimal and user friendly and a source code designed to be synchronized and handled in order for multiple threads to be processed coexisting with the main Android User Interface without causing issues even if the protocol is tested on singled cored-single threaded devices with low system capabilities.

Various tools were used for the implementation to occur including Netbeans, Ecclipse and Android Studio with fully updated libraries and AVD, ADB and API packages. Google Android Studio proved to have the most optimal Java compatible environment to use in the end, due to the ability to compile the code, generate and boot an emulated device and install and run a developed application in relatively very fast speed and low RAM and CPU power consumption.

There were unavoidable general issues generated by the main characteristics of the Android Development Environment such as the lack of automatic syncing for some source files during importing the project leading to manual importation of these files in order to avoid confusion and the failure of the compile to do the tasks needed.
Code Content and References

MAINACTIVITY.java

includes:

start(): sends a signal to the handle to start the application's main thread synchronized with the main Android OS UI thread.

stop(): sends a signal to the handler for a main thread interruption

settings(): leads to the SETTINGS activity where the user sets the variable values for the according connection

loadsettings(): loads the values from the file settings.txt and prints them to the UI console

exit(): exits the application

other helpful methods:

doStart(): prints a UI console message showing that the start button was correctly pressed

doStop(): prints a UI console message showing that the stop button was correctly pressed

showsettingscontent(): reads and prints the content of the settings.txt file on the compiler console

printall(): prints the variable values (can be seen via compiler's console)

activity_main.xml

includes:

A loading animation showing that the main Android UI thread is synchronized.

If the animation freezes, it is a sign that either the Android OS is crashing or there are hardware problems leading to a lack of synchronization between the application's threads and the Android OS main UI thread.

A Start button, that calls the start() method when pressed

A Stop button, that calls the stop() method when pressed

A Settings button, that calls the settings() method when pressed

An Exit button, that calls the exit() method when pressed

A Console text-view, which prints some basic useful messages on the UI.
**SETTINGS.java**

The class includes all the "get" methods for data encapsulation and is basically used in order to get all the variable values that are needed in order to start the application's main thread.

**Also includes:**

- **saveall(View view)**: Once the button "Save changes" is pressed, the method reads all the fields as Strings, it prints the variable values on the UI console and makes sure that all values are going to be saved by calling the saveall() method.

- **saveall()**: After the user has set all the settings needed in the fields, the method saves all the values in a settings.txt file.

- **goback()**: returns the user to the main activity

**Other helpful methods:**

- **showsettingscontent()**: prints the content of settings.txt (can be seen via compiler's console)

- **readFile()**: returns the content of settings.txt into a String

**activity_settings.xml**

**Includes:**

- **A loading animation** showing that the main Android UI thread is synchronized.

  If the animation freezes, it is a sign that either the Android OS is crashing or there are hardware problems leading to a lack of synchronization between the application's threads and the Android OS main UI thread.

- **The fields** needed in order to set the variable values vital to establish a proper connection according to the protocol.

- **A back button**, that calls the goback() method when pressed

- **A Save changes** button, that calls the saveall() method when pressed

- **A Console text-view**, which prints some basic useful messages on the UI.

**UIHandler.java**

The UIHandler class handles the synchronizations between the Android OS main UI thread and the threads needed in order for the application to run starting with the main class thread.

The handler is basically called when the start button is pressed at the MAINACTIVITY and uses data encapsulations (get methods) to transfer the imported values to the main thread.

The main role of this handler is to either start the main thread or interrupt it depending on how the user wishes to handle the connections.
Logger.java

TxtLogger.java
These classes were not edited in any way by implementing the Android version.

Logger.java is used to output log messages in Strings streamed out on console and/or a file

TxtLogger.java is used to output log messages into a predefined output stream on the file
sensor_logs.txt

SensorMain.java
This was originally the main class that starts the sensor's side of the application.
It is transformed into a thread in order to be easier handled.
The class contains get methods and a basic constructor in order to get the values set as inputs and
use them properly in the run() method of the thread.
Once the socket is made a connection request is being repeated until the sink is found.
Once the sink is reached, a message is send to the sink including the sensor's basic information and
then a thread from the class CommHandler is started in order to start ending data

CommHandler.java
The android-studio compiler didn't allow a specific IP address declared in order to bind the
datagram socket needed to start sending the data packages, but using the local port is enough for the
purpose.
When the connection status is optimal, a thread from the class DataSender is started in order to
allow packages to be sent via the DatagramSocket.

Configuration.java
This class contains a list with the types of messages used from other classes

DataPacket.java
This class was not edited for the Android version.
It is used in order to create the packets needed to send data to the sink.

DataSender.java
The class is edited in order to be able to access the input values needed in order to transfer data to
the sink. Data loss and delays are handled properly and accordingly.
Using the Android Application

In order to use the application on an Android device the user has to download and install the app.apk file on the available device and then copy the file “settings.txt” in the SDcard directory of the device.

It is strongly advised that the minimum system requirements have to be taken under consideration before installing and using the application.

When running the application the user is give the chance to work on 2 simple activities.

The first activity is the main menu. The user is introduced to a basic set of buttons accompanied by an informative UI.

The UI background includes the application's version, a continuously loading animation indicates that the UI is not frozen and a console that prints basic useful information.

There are four basic buttons.

The **Start** and **Stop buttons** start or interrupt the main thread of the application. But in order for these to work, the user has to enter the settings menu by pressing the **Settings button** in order to enter the information needed to establish a connection with the Sink and then come back to the main menu and start running the connection and data transfer tests.

When entering the Settings the user is guided through a menu with text fields that need to be filled with the correct information including the IP and Port of the Sink, the IP and local Port of the Sensor, the name and type of the file that is going to be sent, a control Port that will be used during the data transfer, chunkSize and timeout limit values and an ID.

Once the user fills all this information, the **Save changes button** should be pressed for these to be stored and then used during the experiment.

By pressing the **Back** button the user goes back into the **main menu**.

A [detailed user manual](#) exists in the form of URL presented as part of the project.
The logic behind the experiment

In a Sensor network emulated by Android devices running the application simultaneously, the Sink automatically receives data the moment it gets within each Sensor's reach.

At first the Sink, which is moving through the network, is constantly waiting for a connection to occur with any of the sensors.

Figure 5: Sink entering a Sensor network of Android devices
During a hypothetical scenario of an experiment, each of the Android devices running the application has a signal range and will only be aware of the Sink the moment the Sink is within the reach of any of them as shown in Figure 5. The reach of the sensor can be simulated through setting up wifi network signals. For every Sensor signal reaching the Sink, a new connection is being established as shown in Figure 6. Then the synchronized Android threads run applying to the protocol's architecture and all the actions take place while obeying to the exact five steps described above.

Figure 6: Sink and Sensor communicating
Resources used for the project:

Tools and compilers used:
- Google Android Studio
- Eclipse
- Oracle Netbeans

Java and API libraries from oracle.com

Project includes
- source code
- .apk installation file
- html manual
- pdf report

Android Application system requirements:
- Android OS 2.2 or higher
- CPU: 400MHz or higher
- RAM memory: 4 MB or higher
- internal memory: 3 MB or higher
- SDcard memory: 3MB or higher

REFERENCES

[1] C. Stais, G. Xylomenos and Giannis F. Marias, "Sink Controlled Reliable Transport for Disaster Recovery"