UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Computer Science Curriculum

Karl Puusepp

ICD systems' remote control and monitoring tool for Android

Bachelor's Thesis (6 ECTS)

Supervisor: Margus Niitsoo, PhD

Tartu 2014

ICD systems' remote control and monitoring tool for Android

Abstract:

This thesis describes how an Android application for monitoring and controlling automated control systems developed using ICD Software's Control Design Platform was developed.

Keywords:

Automated control systems, Android, Java

ICD kontrollsüsteemide seire- ja juhtimisrakendus Androidile

Lühikokkuvõte:

Selles bakalaureusetöös kirjeldatakse Androidi mobiilirakenduse loomist, mis võimaldab ICD Software'i Control Design Platform raamistikuga loodud automaatjuhtimissüsteemide olekut kuvada ja muuta.

Võtmesõnad:

Automaatjuhtimissüsteemid, Android, Java

Table of Contents

1 Introduction

CDP (Control Design Platform) is a framework and middleware layer developed by Norwegian-based ICD Software AS, designed for easily setting up and configuring reliable high-performance control systems independent of the operating system.

CDP runs a network of interconnected applications and controllers, which interact via real-time signal transmission and processing. Such systems are deployed marine vessels across the world, where each controller is responsible for a subsystem of the ship - monitoring sensors, adjusting hydraulics, logging data, handling user input and so forth.

Over the years CDP has grown into a complex system. Accessing it externally can only be done via an application already in the network or through a cumbersome web interface, which limits extensibility and remote servicing. This is the reason why the company is developing a lightweight, flexible and portable interface called StudioAPI.

The aim of this thesis is to describe this interface and exemplify how it could be implemented and used. The first part of this paper details how the client-side implementation of this interface is written in a Java library. The second part describes how this library is can be used for developing a simple proof-of-concept mobile application on the Android platform, which can connect to a CDP system, display its structure and display and change remote signal values.

2 Control Design Platform

2.1 CDP Overview

Overview

CDP (Control Design Platform) is a platform for designing, developing and deploying automated control systems. It offers a C++ framework, prebuilt libraries for different signal IO protocols, hardware driver wrappers and various tools for designing systems and generating code based on those designs. CDP core library acts as a middleware layer which abstracts OS-specific components like sockets, threads and timers and runs user-defined components on top.

Systems created using CDP consist of applications (binaries), which can discover and interact with each other over a local network. Application behaviour is described by components - state machines that are run by the system at frequencies configurable up to 1000 Hz. Components contain digital signals and properties that can be routed within the application as well as across the network.

CDP aims to make creating and configuring reliable distributed control systems as simple as possible. CDP components are configured almost entirely in XML (eXtensible Markup Language). While it is possible to build a fully automated system using XML and tools provided by CDP with almost no C++ knowledge at all, more complex behaviour and custom components do require writing code [1].

CDP systems do not require any special hardware to run. Applications can be developed and run on ordinary consumer PC's either in Windows or Linux (ARM support is currently unofficial), although they require higher privileges for true real-time performance. Inter-application communication can use standard networking solutions. All this makes developing custom control systems affordable and easy to learn for most developers.

Although not limited to any specific industrial sector, most ICD Software customers use CDP in the offshore industries where the performance, reliability and redundancy features of CDP are important. CDP-driven control systems are installed on marine vessels and are responsible for navigation, propulsion, tank levelling, anti-roll systems and more. CDP controls large motion-compensated gangways, cranes and helicopter decks deployed on ocean-going ships, allowing them to safely operate even in harsh weather conditions [2].

Example system

CDP allows connecting various hardware and software components easily. An example would be a system servicing a large industrial crane (in later sections, this system will be used for the purposes of exemplification).

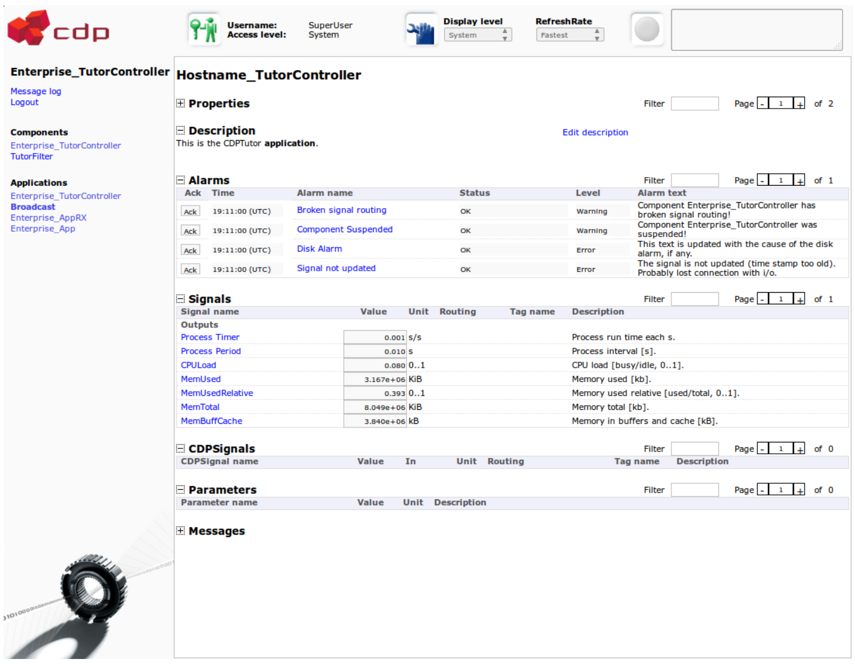
One application (referred to as “Crane” from now on) in the example system has a single component that processes data from the crane’s hardware sensors and emits the current absolute crane position as an output signal.

The second application (referred to as “Logger”) has a component which connects to this signal and calculates it into a relative value, as well as a component managing an SQLite database which logs this relative value at a configurable interval. This database component can be used in other components to get an overview of how often the crane spends in a given position.

2.2 The need for StudioAPI

StudioAPI is part of an ongoing project to consolidate CDP tools into a powerful IDE (Integrated Development Environment) called CDP Studio which could be used to design, compile and test an entire control system. For this to work seamlessly, the IDE would have to interact with a CDP system at runtime, browse its structure, display and plot signal values, change the configuration etc.

At the time of this writing, there are two solutions that offer graphical interaction with CDP. The first is through a web page served by a WebServer component within CDP (Figure 2.1) [1]. This is not a very effective way of communicating real-time data however, as the repeated client-side polling and web page generation is performance-heavy for the application. It also has to be manually updated for each new functionality that becomes available in CDP.



Enterprise_TutorController
Message log
Logout

Components
Enterprise_TutorController
TutorFilter

Applications
Enterprise_TutorController
Broadcast
Enterprise_AppRX
Enterprise_App

Hostname_TutorController

Properties — Filter — Page 1 of 2

Description — Edit description
This is the CDPTutor **application**.

Alarms — Filter — Page 1 of 1

Ack	Time	Alarm name	Status	Level	Alarm text
Ack	19:11:00 (UTC)	Broken signal routing	OK	Warning	Component Enterprise_TutorController has broken signal routing!
Ack	19:11:00 (UTC)	Component Suspended	OK	Warning	Component Enterprise_TutorController was suspended!
Ack	19:11:00 (UTC)	Disk Alarm	OK	Error	This text is updated with the cause of the disk alarm, if any.
Ack	19:11:00 (UTC)	Signal not updated	OK	Error	The signal is not updated (time stamp too old). Probably lost connection with i/o.

Signals — Filter — Page 1 of 1

Signal name	Value	Unit	Routing	Tag name	Description
Outputs					
Process Timer	0.001	s/s			Process run time each s.
Process Period	0.010	s			Process interval [s].
CPULoad	0.080	0..1			CPU load [busy/idle, 0..1].
MemUsed	3.167e+06	KiB			Memory used [kb].
MemUsedRelative	0.393	0..1			Memory used relative [used/total, 0..1].
MemTotal	8.049e+06	KiB			Memory total [kb].
MemBuffCache	3.840e+06	kB			Memory in buffers and cache [kB].

CDPSignals — Filter — Page 1 of 0

CDPSignal name	Value	In	Unit	Routing	Tag name	Description

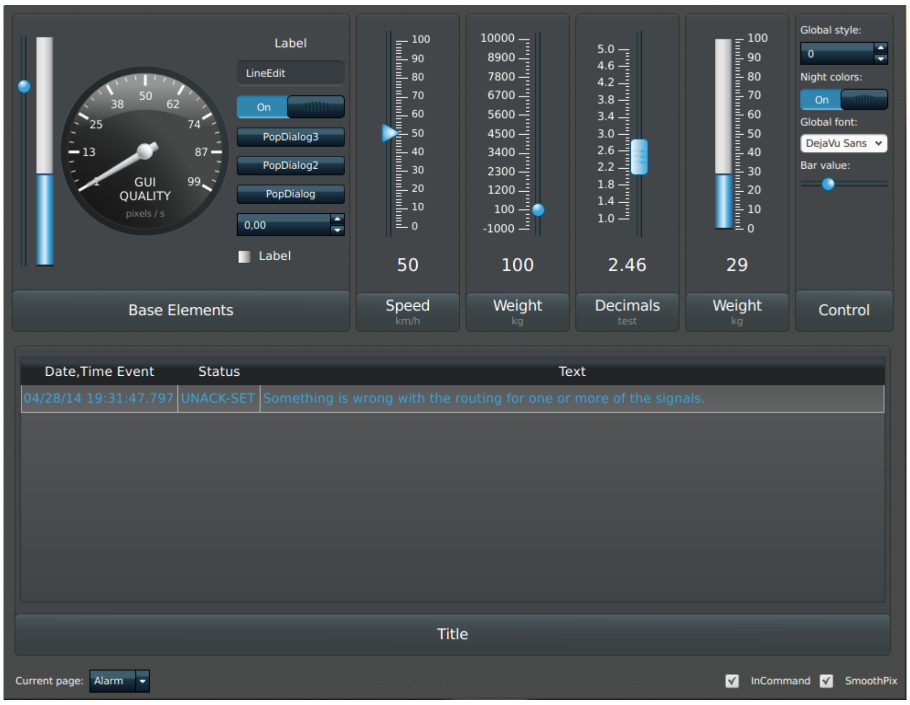
Parameters — Filter — Page 1 of 0

Parameter name	Value	Unit	Description

Messages

Figure 2.1. WebServer interface connected to a CDP Application

The second solution is to use CDP2Qt. This library ships with CDP and allows displaying various CDP objects like alarms, signals and properties in Qt application framework widgets such as labels and lists (Figure 2.2). This is how GUI front ends have interacted with CDP thus far - applications run the Qt framework (which handles the user interface) in the main thread and a full-fledged CDP application in another thread that serves the custom widgets [3].



Date,Time	Event	Status	Text
04/28/14 19:31:47.797		UNACK-SET	Something is wrong with the routing for one or more of the signals.

Figure 2.2. CDP2Qt widgets demo application that ships with CDP

Obviously running a CDP application alongside a fully-featured IDE incurs considerable overhead, not to mention adding new features to the framework would require creating new widgets for the IDE, so CDP Studio required a new, reliable, lightweight and flexible API (Application Programming Interface) for interfacing with any CDP system both locally and over the Internet. StudioAPI was designed to be the new standard "window to CDP".

2.3 StudioAPI design

StudioAPI is a client-server protocol specifically designed by ICD Software for external access to a CDP system at runtime. The StudioAPI server runs as a regular component in newer CDP applications, asking the rest of the application for its structure data via a node data interface that all objects implement. It then serialises the data and sends it to clients, which are completely independent of CDP and can be written in almost any major programming language.

The client-server communication of StudioAPI depends on two key technologies. It uses WebSocket for transport and Google Protocol Buffers (Protobuf for short) for data description. WebSocket offers a reliable and lightweight full-duplex channel over a single TCP connection and Protobuf provides effective and portable serialisation of data.

WebSocket

WebSocket is a protocol used for bidirectional data transmission over a single TCP connection. It was standardised by the Internet Engineering Task Force in 2011 and is currently implemented in all major web browsers. While it is mainly designed for use in web applications, it is not limited to this domain and can be used elsewhere [4].

Because WebSocket allows two-way communication, it's well suited for pushing periodic updates (such as signal values) or unexpected events (such as alarms) from the server without the client explicitly having to poll for them. These features are hard to implement with traditional technologies while maintaining high performance and low overheads. WebSocket also offers splitting messages over multiple packets and SSL encryption. It's these qualities that made it the transport of choice for StudioAPI. StudioAPI server currently uses an LGPL-licensed WebSocket library called libwebsockets, which is written in C.

Protocol Buffers

Protocol Buffers are Google's platform-neutral mechanism for serialising custom data structures. This data is defined in Protocol Buffers' own interface description language (IDL) as messages - classes with required, optional and repeated fields. Google's protoc compiler can then generate C++, Java and Python source code based on this description (many other languages also have third-party support). The generated code turns the messages into native objects, which can be constructed from and serialised, to binary data.

Protobuf technology has many benefits; the most important are the effective serialisation of data, easily implementable forwards and backwards-compatibility between different description versions and very low overhead for optional and repeated message fields [5].

As both WebSocket and Protobuf technologies are open and portable, the client-side implementation of StudioAPI can be written in almost any major language without much effort. This paper exemplifies the process of writing one such implementation in Java and demonstrates its usage in an Android mobile application.

2.4 Protocol description

The main description of the StudioAPI protocol is written in the aforementioned Protobuf IDL. The initial design was described by ICD Software in 2013. It has seen smaller modifications since then and is not yet finalised at the time of this writing, but the existing principles are expected to remain the same.

The CDP system tree

The protocol maps the hierarchic CDP system structure to a tree of generic nodes (described in the PBNode Protobuf message type). A single node can correspond to an entire application or simply the input routing address of a single integer-type signal. This depends on the node's type name string and location in the tree. Each node is assigned a unique ID by the CDP runtime, which is used to identify nodes between server requests and responses.

The tree structure is polled using these node IDs. The exception is the initial structure request, which does not specify a node ID (taking advantage of Protobuf messages' optional fields feature). A structure response from the server contains the node for which structure was polled as well as all its immediate child nodes. Further structure can be polled for each received child node separately via their node IDs.

A simplified example of the node tree is in Figure 2.3. Actual CDP systems can have hundreds of thousands of nodes, since every signal has multiple properties and every property has multiple settings. This means client implementations should limit the depth of structure polling as much as possible to keep their memory footprint small and limit the server load.

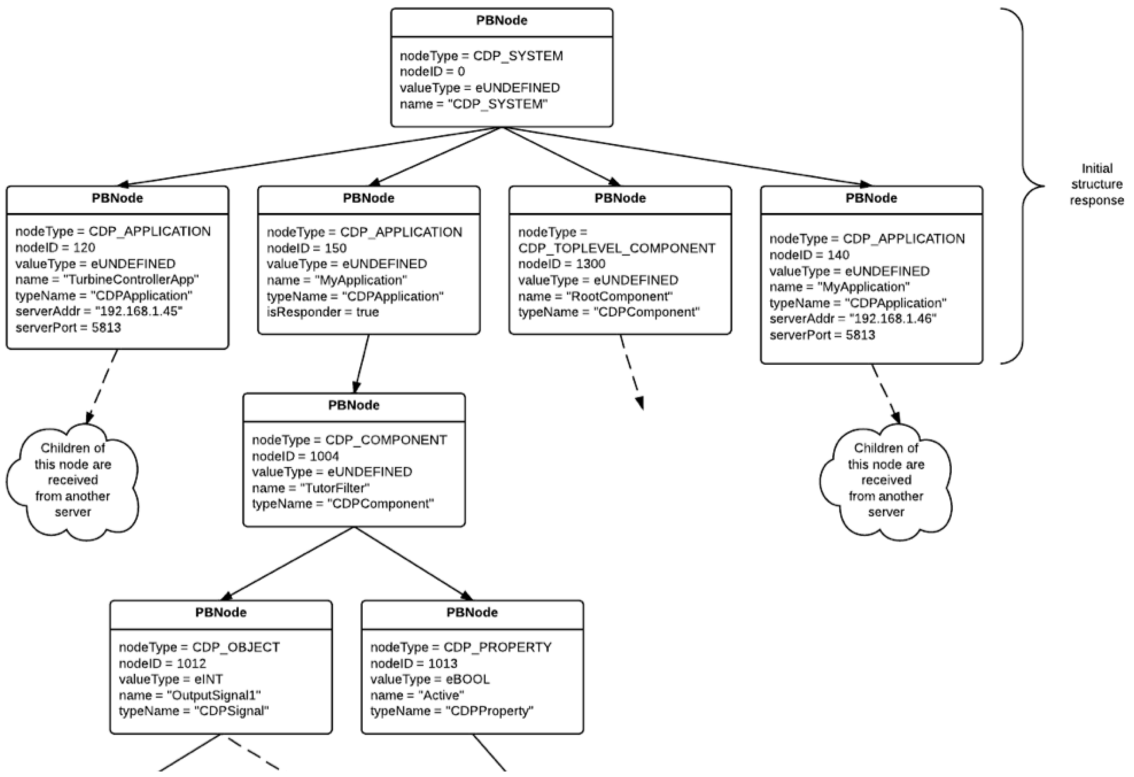


Figure 2.3. Part of an example node structure for a CDP system.

The initial structure response is almost the same for all applications in the network, meaning that they return a CDP system node (with a node ID of zero) as well as application nodes for the entire network known to them. Each server also returns any top-level components belonging to that application. These are not separate applications but components, which reside under the root level similarly to applications.

Other features

For nodes that have a defined value type, single value requests and value change subscriptions can be sent. The protocol encapsulates all 12 possible value types for a node as a variant value (Table 3.2). This variant also has a timestamp in the form of a double, which counts the number of seconds from the CDP application's start time. The variant is equipped with a node ID and sent as a response to structure requests. Variants can also be used for setting node values on the client-side.

Lastly, the protocol describes a way of subscribing to changes in system structure and how these events are reported. Subscriptions can be made to any node with a supplied depth. A depth of zero subscribes to the entire subtree, while a depth of one only subscribes to the event of the node itself getting deleted. Structure change events are reported as a PBNode of the new structure along with a timestamp and the author of the change.

Example

To change the logging interval for the database component described in section 1.1.2, the client would have to send the following protocol buffer messages to the "Logger" application's StudioAPI server:

1. An empty structure request to get the node ID of the "Logger" application.
2. A structure request for the "Logger" application's node ID to get the ID's of its component nodes.
3. A structure request to the component responsible for the SQLite database to get the node ID of the logging interval property.
4. A structure request to the component responsible for the SQLite database to get the node ID of the logging interval property.

2.5 Porting StudioAPI to Java

StudioAPI enables a whole ecosystem of different clients to interact with CDP. They can be built in many languages and run on most modern platforms. Java is one of the most popular platforms in use today, so providing a reference implementation of the client for it would benefit existing solutions developed by ICD's customers as well as allow new solutions to be created with little effort.

The following sections detail how this reference implementation was created and used in a simple mobile application which allows project engineers to get an overview of a CDP system.

3 StudioAPI Client in Java

The following paragraph describes the principles employed for implementing the StudioAPI client-side protocol in a simple Java library.

3.1 Technologies used

The first consideration in designing the library was choosing a WebSocket implementation. There were two important requirements for it:

1. It must be compatible with libwebsockets (written in C) used by StudioAPI server.
2. It must be portable and work in Android as well in any desktop Java environment.

The second requirement turned out to be harder to fulfil than expected. While there are popular WebSocket implementations for both Android and Java EE separately, the number of portable options is low. The most solid candidate at the time was a library called Java-WebSocket, which fit both key requirements.

Apache Maven was chosen as the build automation and dependency management tool. Maven simplifies build automation considerably, as it uses conventional source and test folder hierarchies for defining project structure instead of explicit build commands. It's also capable of including dependent libraries from the Internet with minimal configuration. In 2013, Maven was the most popular Java build tool according to [6].

Another bonus of Maven is its compatibility with Gradle, the build system used for Android projects, and the fact that both Google Protobuf and Java-WebSocket are already available in the official Maven repository. This means that Maven can automatically resolve those dependencies whenever StudioAPI Java client is included in a Maven or Gradle project.

3.2 Library interface

The StudioAPI client interface is designed to be as simple to use as possible. The bulk of the functionality is in two classes:

1. The Client class is responsible for initialising the connection(s) and holding the root node of the cache.
2. The Node class is the cache storage element. It has accessor methods for all its main attributes as well as other cached child and parent nodes.

Nodes also expose an asynchronous interface for making structure requests and both value and structure subscriptions. These requests, once resolved, will call a callback that the user must supply with the request. For structure requests where the data had already been received from the server, the request is resolved and user notification will happen instantly, no data is sent to the servers.

3.3 Library implementation

As described before, a CDP system's first-level nodes (CDP Applications) all correspond to separate connections; therefore it's the responsibility of the library to hide these as an implementation detail. As a result, all user requests are either done to the main client instance or to the cached nodes, other public classes and interfaces are mainly used for event callbacks.

The client caches all the structure data it receives and exposes references to that cache. If cache data is added or removed, appropriate user callbacks are called. The client avoids making any requests to nodes that were not explicitly requested by the user, with the exception of the top-level structure (application and top-level component nodes under a shared system node), which must be cached first.

The initial WebSocket connection can be opened to any StudioAPI server in a given system. This server responds with the top-level structure of its own CDP application, as well as the root nodes of other applications in the network. These nodes contain connection parameters that are used to open secondary connections.

Once all connections have been created and the top-level structure has been cached, the appropriate user callback is called to signal successful client instantiation. The user can then access the cache and make new requests to the nodes in it.

Internally, each node has a reference to a connection handler that holds a single connection to an application. When an asynchronous value request, structure request or a subscription is made to a node, the node forwards it to its responsible handler, which in turn forwards it for serialization and sending (if the request was valid and the necessary data wasn't already cached). The handler creates a pending request object and returns it to the user, while keeping a reference to it in memory.

If new data is received from a WebSocket connection, it is deserialised and sent to the responsible connection handler. The handler updates its cache structure and marks any request objects that were expecting the received data as resolved. If the connection that the handler uses was dropped, it removes the subtree of the general cache that it was managing and the user is notified that this part of the structure is no longer valid.

Threads in StudioAPI

The chosen WebSocket implementation, Java-WebSocket, uses threads extensively to provide non-blocking event-driven IO. Data writes are sent to a concurrent queue, which is emptied by another thread handling socket writes. Data is received in the WebSocket client thread, which calls various user callbacks when events have occurred (Table 3.1).

Table 3.1. Threads used in Java-WebSocket

Thread	Usage
Original thread	The WebSocket client is created on this thread.
WebSocket events	Constantly polls the external server and triggers events.
WebSocket write	Pulls data from write queue and sends it to server.

As WebSocket is asynchronous by design, any client-side implementations of StudioAPI should follow the same principles. This means data is provided to the user in the form of callbacks that are called from the context of an event loop. A similar design is employed in many UI frameworks, such as Java Swing GUI framework, the Qt framework as well as Android itself.

This means that the Java client needs to run its own event processing method. Although it mainly receives events from Java-WebSocket, it would constrain the API to build it on top of the limited event callbacks and thread scope of the transport layer. Since the layer doesn't offer any "default action" callback to write a custom event loop on top of (in case events need to be processed but no input has been received from the server), the client library runs its own event loop in a thread separate from Java-WebSocket.

To help simplify library usage, the main client class both implements the Java Runnable interface as well as exposes its main event processing method. This means that the client can be either started in a new thread or, alternatively, the process method can be added to an already existing event loop like the main Looper instance found in Android application using periodic timers.

Such a design has the benefit of offloading thread synchronisation responsibility to the user while also making it completely optional. When launching the client in a separate thread, the user is responsible for writing all event handling into the supplied callbacks, but running the client from an already existing thread removes the need for thread-safety mechanisms and requests can be safely made from other methods in the user's event loop.

Type system limitations

Although Protocol Buffers are equally supported for both Java and C++ applications, there are problems with making C++ and Java type systems compatible. For most use cases, these problems are trivial, but CDP uses templates extensively in internal code and offers 11 different value types for CDP signals and 12 types for CDP properties and CDP settings. Some of these types are not officially supported by Protobuf and many of them are not supported by Java.

Table 3.2. Corresponding value types in C++, Protobuf and Java

C++ value type in CDP	**Protobuf equivalent**	**Java equivalent**
double	double	double
float	float	float
int	sint32	int
long long	sint64	long
bool	boolean	boolean
std::string	string	String
unsigned int	uint32	-
unsigned long long	uint64	-

char	-	-
unsigned char	-	-
short	-	short
unsigned short	-	-

In the C++ implementation of StudioAPI, all node values are wrapped in a type-safe variant class which holds a value type identifier (enum), a pointer to the value itself and an optional timestamp corresponding to how many seconds after application start the value was assigned. The variant class' constructor is defined for all 12 supported value types and the correct enum is deduced implicitly. When setting a value to a node, the variant's type identifier is checked against the node's value type. When sending variants over the network, unsupported value types (signed and unsigned chars and shorts) are safely casted to larger integer fields.

In Java, many of these value types are not available, meaning variant instances must be constructed with an explicit type identifier. As setting node values does not happen often (in contrast to value updates which may happen hundreds of times per second), the library only permits the user to construct values by supplying the type identifier and a string which is parsed accordingly. For encapsulating unsigned 16-bit, 32-bit and 64-bit integers the library follows the same logic as Protobuf - they are represented as their signed counterparts with the top bit stored in the sign bit [8].

4 Using StudioAPI in Android

The created StudioAPI Java client can be used with any standards-conforming implementation of the Java runtime. This paragraph describes how the library is used in a mobile application called CDP Remote that displays CDP component structure, lists signals, rich signals and properties of each component with their latest values and allows setting new values to them. While it does not take advantage of the full feature set of the library, it can be a useful tool for monitoring a CDP system remotely and for getting an overview of its state.

4.1 Why Android?

Android was chosen as the development platform for two reasons. The first is simply because the Java implementation can be natively run on Android already. The second consideration is more practical - Android is available on a wide range of hardware. CDP is mainly used in industrial settings and Android tablets and phones can be purpose-built to withstand dropping, water or dust.

4.2 Choosing an Android version

Android is considered to be a much more fragmented ecosystem than its competitors (Figure 4.1). Although Google's efforts and time have improved the situation, the problem still remains. Different versions of Android 2.3 (codenamed Gingerbread) are still used on 19% of all devices, which is a considerable share of the market. However, Gingerbread is essentially not used on tablets and extending support for it would limit the number of available toolkit features greatly.

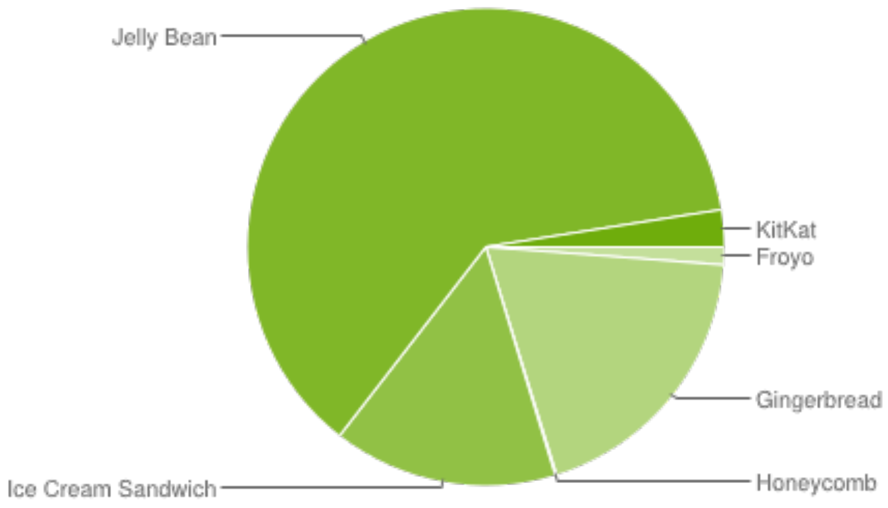


Figure 4.1. Google Play Android version statistics over 7 days before March 3rd, 2014 [7].

Android 3.0 (Honeycomb) was the first version of the OS specifically aimed at tablets, but couldn't gain any traction on the market. Honeycomb is currently run on only 0.1% of all devices using Google Play. Android 4.0 (Ice Cream Sandwich) was the first serious attempt to unify the phone and tablet versions of the OS. It introduced a unified UI framework and made it simpler to target multiple devices with a single application. This is also the reason why it was picked as the lowest targeted version, as CDP Remote should offer a consistent experience on both tablets and phones, but also run on most hardware currently available.

4.3 Graphical user interface

Android Fragments

The GUI (Graphical User Interface) of CDP Remote follows a simple utilitarian design and uses Android Fragments for its main view. Fragments were introduced in Android 3.0 (Honeycomb) to help create more flexible and reusable design elements.

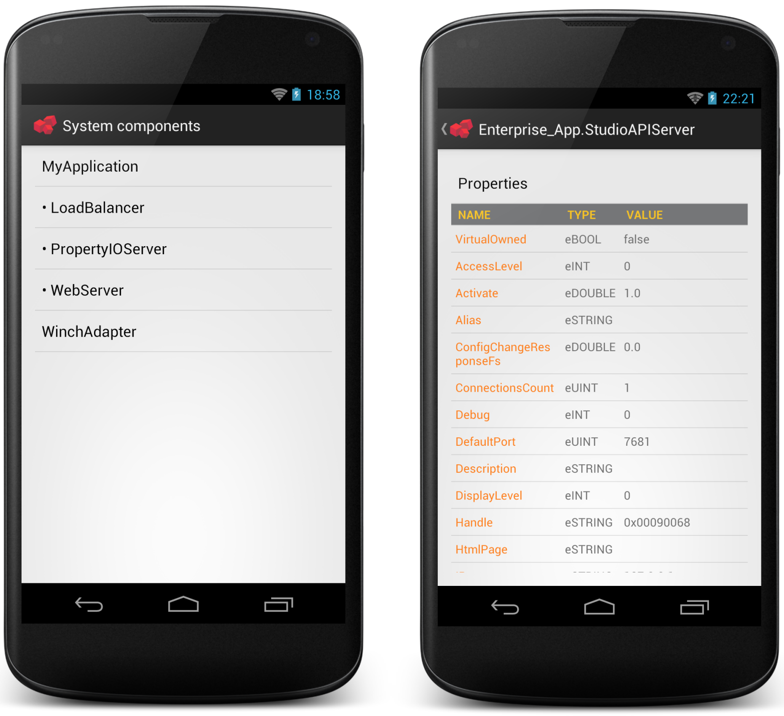
In Android, the current view visible on screen generally corresponds to a single Activity - an object that handles the View drawing and state, as well as transitions to and from other Activities (which may live in different applications). Fragments allow much of the Activity behaviour to be extracted into a smaller, separate reusable container. An Activity may compose of a single Fragment or display multiple Fragments at a time when a larger screen size permits it.

A general design pattern that uses Fragments is a master-detail view. On smaller devices (mainly smartphones) it consists of two linked Activities. The first is the master view -

usually a list of data elements. Clicking on an item opens a detail view, which displays the content or detailed data for the element. On larger devices (mainly tablets) the separate master and detail Fragments are combined into a single Activity. This design makes navigation easier for the user and uses the additional screen real estate in a more optimal manner [9].

Design description

When starting CDP Remote, an Activity asking for the server address and port is displayed. Once the connection has been successfully established, the user is transitioned to a master-detail view. The master view lists all CDP applications and components in the system. Clicking on a component opens the detail view where all properties, signals and rich signals (otherwise known as CDPSignals) of the Component are listed along with their most common fields. In a smartphone, these are displayed in a separate Activity. The user can return to the listing by using the hardware or software “back” button (Figure 4.2). On a tablet, the two views are combined (Figure 4.3).


18:58

System components

MyApplication

- LoadBalancer
- PropertyIOServer
- WebServer

WinchAdapter

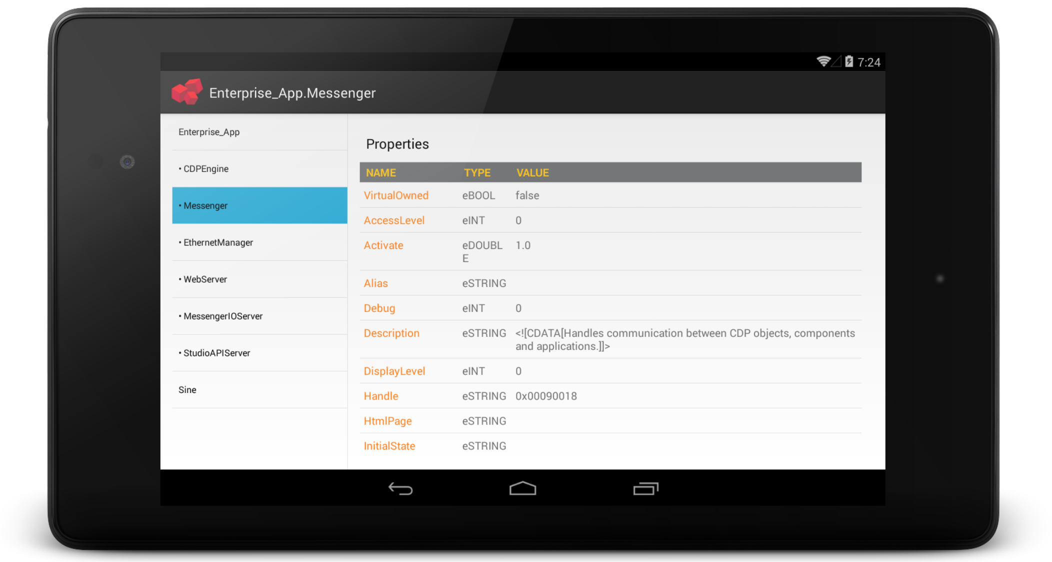
22:21

Enterprise_App.StudioAPIServer

Properties

NAME	TYPE	VALUE
VirtualOwned	eBOOL	false
AccessLevel	eINT	0
Activate	eDOUBLE	1.0
Alias	eSTRING	
ConfigChangeResponseFs	eDOUBLE	0.0
ConnectionsCount	eUINT	1
Debug	eINT	0
DefaultPort	eUINT	7681
Description	eSTRING	
DisplayLevel	eINT	0
Handle	eSTRING	0x00090068
HtmlPage	eSTRING	

Figure 4.2. CDP Remote user interface on a Nexus 4 smartphone.



Enterprise_App.Messenger

- Enterprise_App
- • CDPEngine
- • Messenger
- • EthernetManager
- • WebServer
- • MessengerIOServer
- • StudioAPIServer
- Sine

Properties

NAME	TYPE	VALUE
VirtualOwned	eBOOL	false
AccessLevel	eINT	0
Activate	eDOUBL E	1.0
Alias	eSTRING	
Debug	eINT	0
Description	eSTRING	<![CDATA[Handles communication between CDP objects, components and applications.]]>
DisplayLevel	eINT	0
Handle	eSTRING	0x00090018
HtmlPage	eSTRING	
InitialState	eSTRING	

Figure 4.3. CDP Remote on a Nexus 7 tablet computer.

Fragment lifecycle optimisations

A single active connection to a StudioAPI server can be very taxing to the system with multiple active value subscriptions sending dozens or even hundreds of TCP packets every second. Although the StudioAPI Java client already caches all structure information and most recent values it has received, the API users should also avoid maintaining value subscriptions whenever possible.

The Android framework is heavily event-based and both Activities and Fragments have lifecycles with appropriate state transition callbacks. When a Fragment goes out of view, it is destroyed and an appropriate subclass method is called. So every fragment keeps track of all the values that are asked of it, subscribes to them and unsubscribes once the view goes out of scope.

Example usage

When connecting to the system described in section 2.1, the master view lists both “Logging” and “Crane” applications as well as the components under each application. The user can navigate to the crane hardware component and see the current absolute crane position changing in real-time in the detail view.

The user can also navigate to the component responsible for SQLite logging and see the logging interval property. Clicking on the property’s value field will allow the user to set a new interval value. If multiple users are changing interval, the value updates will be displayed to both of them. If the SQLite database is nearing a capacity limit, an alarm may be displayed under the property listing of the component.

4.4 Android framework limitations

As this was the author's first foray into Android development, there were some aspects of the framework that came as a surprise and required awkward workarounds.

Firstly, transitioning from one Activity to another is done through Intents. When some event occurs which should transition away from the current Activity, the Activity creates an Intent, sets the class of the new Activity to be transitioned to and packages any additional data that it wishes to pass on. The event loop later creates an instance of the new class and calls its creation callback, where the packaged data is available.

However, this data can only be passed on as plain value types or serialised data, meaning everything is reconstructed in the new Activity and no references are kept. This is obviously a problem when trying to pass shared objects like open sockets or specific references to cached tree nodes, both of which are required by StudioAPI, so the recommended solution is either to inherit from the Application class and store global data there or use static singleton instances. For the sake of simplicity, the CDP Remote implements the latter.

The second limitation is the lack of a shrink-to-fit functionality for the standard Android text view. This means that long strings will be broken to multiple lines, sometimes at completely arbitrary points, and setting the text size from code requires more complex conditional branching. This was worked around by testing the program on both a smartphone emulator with a 4.7" display size and a tablet emulator with a 7" display size and picking a sensible default that suited both of those configurations in portrait and landscape orientations.

4.5 Future developments

At the time of writing this thesis, StudioAPI is still being finalised and communication between the server and CDP itself has not been fully implemented, so the created Java library is only as feature-complete as the current protocol and server-side implementation let it. The next key features for the library and Android app are:

- Support for SSL encryption (already supported by the client and server technologies used, but not supported in StudioAPI server yet).
- User authentication (not yet defined in the protocol).
- Plotting signal and property value changes on a graph on Android.
- Customisable table views on Android

5 Summary

The purpose of this bachelor’s thesis was to implement the client-side interface for connecting to automated control systems developed using the Control Design Platform (CDP) framework as a Java library and develop a mobile application for the Android platform called CDP Remote which uses this library.

This thesis details how the StudioAPI protocol abstracts a CDP system’s structure and what technologies are used by it. It describes how the client library implements this protocol, optimises its usage and safely abstracts more complex elements of the type system and multithreading from the user.

The developed mobile application is a useful tool for getting a general overview of a CDP system’s state at runtime and tweaking property and signal values. It has a simple graphical user interface which adapts to both smartphones and tablet devices. It is not yet ready for real-world use however, as the StudioAPI protocol and integration with CDP hasn’t been finalised.

6 References

[1] ICD Software AS (2014, January) CDP System Manual

[2] ICD Software AS, Cases. [Online].
http://www.icdsoftware.no/products/cases (04.05.2014)

[3] ICD Software AS, CDP2Qt V3.5.0.0 User Manual (2013)

[4] Internet Engineering Task Force (2011, December) RFC 6455 - The WebSocket Protocol. [Online].
http://tools.ietf.org/html/rfc6455

[5] Google Developers (2012, April) Developer Guide - Protocol Buffers. [Online].
https://developers.google.com/protocol-buffers/docs/overview

[6] RebelLabs (2014, January) Java build tools. [Online].
http://zeroturnaround.com/rebellabs/java-build-tools-part-2-a-decision-makers-comparison-of-maven-gradle-and-ant-ivy/

[7] Android Developers (2014, April) Dashboards. [Online].
http://developer.android.com/about/dashboards/index.html

[8] Google Developers (2014, April) Language Guide - Protocol Buffers. [Online].
https://developers.google.com/protocol-buffers/docs/proto

[9] Android Developers, Supporting Tablets and Handsets. [Online].
http://developer.android.com/guide/practices/tablets-and-handsets.html (27.04.2014)

License