MASTER DESIGN PLAN
FOR
PTOLEMY ON ANDROID

VERSION 1.0

PREPARED BY TEAM HANDSIMDROID

CARNEGIE MELLON UNIVERSITY

04/28/2011

TABLE OF CONTENTS

1 INTRODUCTION

1.1 REPORT COVERAGE

The purpose of this document is to outline the architectural design of the system to be constructed by Team HandSimDroid for the Bosch Research and Technology Center (RTC) and the Ptolemy Project Team at the University of California at Berkeley through Carnegie Mellon's Master of Software Engineering program. This document includes the high-level design, project decisions and accompanying tradeoffs, quality attributes and scenarios, as well as business and technical constraints for the Android Simulation Tool, User Interface Layout Designer and Ptolemy Server.

1.2 INTENDED AUDIENCE

This document is intended for the client, Bosch RTC, the Ptolemy Project Team, Team HandSimDroid, the MSE Project Mentors, and our instructors in Software Architectures class. It may also be useful to future MSE students, faculty, and staff in becoming more familiar with the scope and architectural design of the project or to serve as a reference point for future studio projects.

The following table describes recommended section for different audience.

	Bosch	Ptolemy Project Team	HandSimDroid Team	Mentors	Architectures Instructors
Section 1 Introduction	✔	✔	✔	✔	✔
Section 2 Project Overview	✔	✔	✔	✔	✔
Section 3 High Level Architecture	✔	✔	✔	✔	✔
Section 4 First Level Decomposition	✔	✔	✔	✔	✔
Section 5 Analysis of the System	✔		✔	✔	✔
Section 6 Appendix A: Experiments	✔	✔	✔		✔
Section 7 Appendix B: Second Level Decomposition			✔		
Section 8 Appendix C: Structural Analysis of Ptolemy	✔	✔	✔		

2 PROJECT OVERVIEW

Bosch Research and Technology Center (RTC) has tasked Team HandSimDroid with building a solution that would allow models developed in Ptolemy, an open-source modeling and simulation tool, to be run in real-time, manipulated, and operated using a user-configured layout on Android-powered devices. Robert Bosch GmbH is a worldwide corporation that designs and develops embedded systems for automobiles. Because of the inherent complexity of modern engine systems and necessary precision to ensure safe operation, Bosch uses model-driven development. These systems are currently modeled by the various business units using a tool called ASCET that is developed by a Bosch subsidiary, ETAS. Though ASCET has sufficient capabilities for their current operations, the Bosch RTC has been researching additional capabilities that could be incorporated into the ASCET toolkit and provide benefit.

Ptolemy, developed by an open-source community primarily at the University of California at Berkeley, provides users with the ability to simulate real-world activity and perform advanced analysis on models. However, Ptolemy needs some additional enhancements to fully support the goals of Bosch RTC.

Unfortunately, the current desktop solution of Ptolemy does not lend itself to more portable applications where the user is on the move, changing model inputs/outputs and parameters, observing actual engine conditions, and providing immediate feedback. There are also limitations to the handheld devices, namely the screen real estate, processing power, battery life, and available memory, particularly tablets and phones, when compared to their desktop counterparts. That being the case, the simulation must provide only the displays appropriate to the end-user and, because the needs can vary greatly between engineers, must be laid out in a highly configurable manner. An effort to support this functionality has been undertaken in the past, but is not fully functional in the Ptolemy tool. With these extensions in place, Bosch RTC will be able to demonstrate Ptolemy running on an Android device with a user-specific layout.

To summarize, the ETAS commercial tool ASCET is used to facilitate model-driven software development of embedded systems. Bosch RTC, in an effort to explorer other potential uses and extension points, has been experimenting with an open-source, concurrency modeling and simulation tool called Ptolemy. This MSE project, which builds on the functionality of Ptolemy and extends its capabilities to another, more versatile platform, will act as a proof-of-concept of functionality that may, if proven to be both useful and feasible, be incorporated into Bosch's commercial tool at some later point.

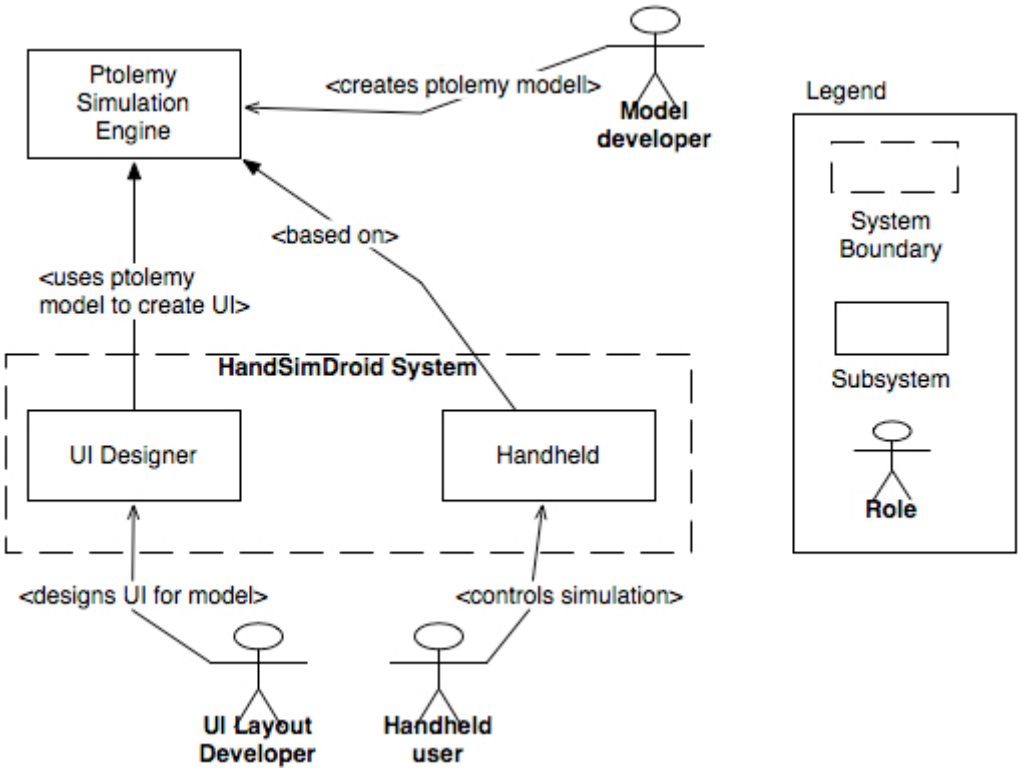


Figure 1: The context diagram of the HandSimDroid project

As described above, Figure 1 shows the context of the envisioned system. The system provides a UI Designer that will enable a user to design a user interface for a model used on the Android handheld. The simulation system on the handheld is based on the model-driven simulation engine of Ptolemy of which we do not intend to change.

2.1 LEGACY SYSTEM

The Ptolemy legacy system has a long history, and is currently developed at the University of California at Berkeley. It supports a model-driven, pipe-and-filter system for model simulation, along with implemented Ptolemy actors (filters in the system) and a visual interface for the model creation and simulation control. Figure 2 and 3 show the communication mechanism of the pipe-and-filter models.

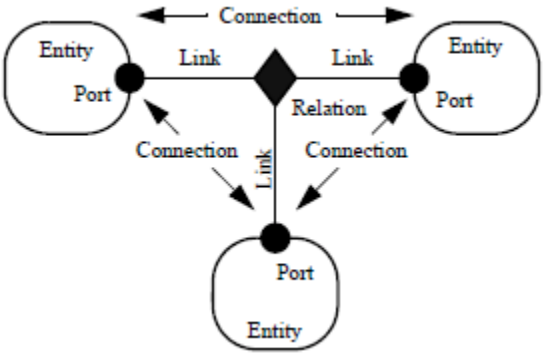


Figure 2 [2, pg. 2]: Entities (actors or filters), links, and the pipe-and-filter architecture within Ptolemy. The entities are linked and can only communicate through existing links connected with relations.

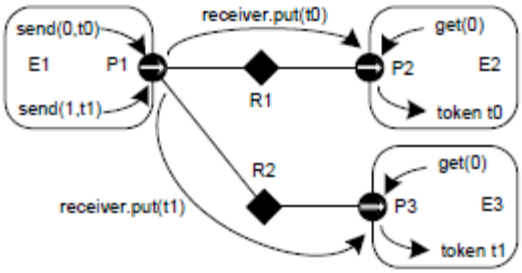


Figure 3 [2, pg. 31]: A sample communication flow of the Ptolemy pipe-and-filter architecture based model. The model uses immutable objects called tokens to transfer any kind of data.

Element	Responsibility
Entity	Entities are the filters in the supported pipe-and-filter architecture. They are responsible for computation within the system. An entity can contain any number of ports.
Port	Port is a point of connection for the entity. It can serve both as output and input, and knows if there's a token waiting for processing.
Relation	The relation is responsible to keep track of links. It controls the creation and termination of links
Token	An encapsulation of a data message. Token serves to identify the message type, size and data boundaries
Link	A link defines an association between an entity's port and a relation.

The Ptolemy architecture was designed to be highly extensible, allowing new entities, links, relations, and ports to be easily added during compile or runtime without impacting high level architecture or implementation of other modules. For the complete description of the architecture, see the up-to-date architecture documentation at http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-29.html.

The architecture of Ptolemy **promotes** extensibility by enabling users to easily create new actors, relations, and links. It promotes configurability by its entity state-modifying mechanism and performance in terms of computational power due to potential parallel computations done by the entities. Flexibility and reuse is promoted through the decoupling of the entities, following the intent of pipe-and-filter architecture systems.

On the other hand, the architecture **inhibits** performance in terms of latency, as tokens have to be parsed and un-parsed throughout the system to facilitate communication. Because it was aimed to be useful in a number of domains, the architecture is also complex.

2.2 Architectural Drivers

2.2.1 High-Level Functional Requirements

The high-level functional requirements of the system can be broken into three very broad categories, each of which is detailed in the addendum to the Software Requirement Specification that can be viewed at http://msesrv4a-vm.mse.cs.cmu.edu/wiki/index.php?title=Artifacts

- The user must be able to run, control, and visualize a Ptolemy simulation on an Android-powered handheld device.
- The team must develop a tool for constructing a customized user interface, particular to a model and user, which can be used by the Android-powered handheld device. This will allow engineers to view only the simulation outputs that are important to them and manipulate the model parameters as necessary.
- The team must implement a framework that will simplify the creation of new display actors capable of operating on Android

2.2.2 Quality Attributes

The Table 1 represents the quality attribute scenarios that have been discovered from the Quality Attribute Workshop (QAW) and have been refined multiple times. These quality attributes were prioritized by the client and have been presented in descending order of difficulty as determined by the team.

Difficulty ranking is defined in the ACDM method and shows the perceived difficulty of implementing the quality attribute scenario in the system. The rating starts with “easy”, the next level of complexity is “challenging”, and lastly “difficult” being the most complicated case.

ID	4
Stakeholder Priority	High

Difficulty Ranking	Challenging
Quality Attribute	Reliability
Source of stimulus	Ptolemy simulation
Actual Stimulus	The handheld user is running a model that experience a Ptolemy simulation error that stops the normal execution
Artifact Affected	System
Environment	Normal operation
Effect of the Action	The handheld application recovers from a run-time error thrown by the simulation engine such as bad parameter in the model or incompatible actors.
Response Measure	The handheld application provides the user with a way to cancel the simulation's execution, return to a default state, and log the error for future debugging

ID	5
Stakeholder Priority	High
Difficulty Ranking	Challenging
Quality Attribute	Extensibility
Source of stimulus	Ptolemy Developer
Actual Stimulus	A Ptolemy developer adds an existing graphical actor to be used for the handheld application
Artifact Affected	System
Environment	Normal operation, compile-time
Effect of the Action	A new graphical actor is added

Response Measure	A new actor is incorporated into the desktop user interface designer and is displayable on the handheld within two (2) person weeks

ID	6
Stakeholder Priority	High
Difficulty Ranking	Challenging
Quality Attribute	Extensibility
Source of stimulus	Ptolemy Developer
Actual Stimulus	A Ptolemy developer adds an existing input actor that pulls data from the connected sensor to be used for the handheld application and incorporated into desktop interface designer
Artifact Affected	System
Environment	Normal operation, compile-time
Effect of the Action	A new sensor can be used on the handheld
Response Measure	The new sensor is incorporated into the desktop user interface designer and handheld application within two (2) person weeks.

ID	7
Stakeholder Priority	High
Difficulty Ranking	Difficult
Quality Attribute	Performance
Source of stimulus	Ptolemy simulation
Actual Stimulus	The handheld user runs the sound spectrum model

Artifact Affected	System
Environment	Normal operation, up to three source sensors and three output sinks (graphs). Samsung Galaxy Tab handheld is running the handheld application with most of the hardware resources allocated to it.
Effect of the Action	The sensor data is captured and fed into the simulation with the order preserved
Response Measure	The latency between the sound captured on the handheld and the subsequent processing commencement on the server must be below 2 seconds per sensor source. The latency between the sound finished processing on the server and the subsequent output on the handheld must be below 2 seconds per sink actor.

ID	8
Stakeholder Priority	High
Difficulty Ranking	Difficult
Quality Attribute	Usability
Source of stimulus	Ptolemy Interface Designer
Actual Stimulus	The Ptolemy interface designer creates a custom user interface for some Ptolemy model using the desktop tool
Artifact Affected	System
Environment	Normal operation, runtime
Effect of the Action	When the user executes a simulation on the handheld device, he/she can view and interact with that model's custom user interface
Response Measure	The user interface on the handheld looks exactly as it was designed and shown in the desktop preview

ID	9
Stakeholder Priority	Medium
Difficulty Ranking	Easy
Quality Attribute	Extensibility
Source of stimulus	New Android version
Actual Stimulus	Version 3.0 of Android comes out
Artifact Affected	System
Environment	Normal operation
Effect of the Action	The user is able to execute the simulation on the handheld device
Response Measure	The layout builder and handheld application support future versions of Android without any code changes

ID	10
Stakeholder Priority	Medium
Difficulty Ranking	Easy
Quality Attribute	Extensibility
Source of stimulus	New Android device
Actual Stimulus	An interface designer is building a layout for a new Android device with different dimensions and sensors
Artifact Affected	System
Environment	Normal operation, runtime

Effect of the Action	The user performs the simulation on the handheld device an downloads the new UI layout
Response Measure	The handheld application works on the new Android Device with no impact on its implementation. Simulation engine is able to use new sensors and the custom user interface meets QAS 8

ID	11
Stakeholder Priority	Medium
Difficulty Ranking	Easy
Quality Attribute	Extensibility
Source of stimulus	New Android device and OS
Actual Stimulus	Version 3.0 of Android comes out with new features or sensors
Artifact Affected	System
Environment	Normal operation
Effect of the Action	RTC can implement new features to the software
Response Measure	No changes are required to the system architecture

ID	12
Stakeholder Priority	Medium
Difficulty Ranking	Easy
Quality Attribute	Maintainability
Source of stimulus	Ptolemy Developer

Actual Stimulus	A Ptolemy developer needs to maintain this system by making a change to the code
Artifact Affected	System
Environment	Normal operation, design/implementation time
Effect of the Action	The developer makes a necessary change
Response Measure	The effort to understand and identify where the change needs to be made is 0 person/days.

ID	13
Stakeholder Priority	Medium
Difficulty Ranking	Easy
Quality Attribute	Reliability
Source of stimulus	Ptolemy simulation
Actual Stimulus	The handheld user runs a model that requires Wi-Fi input
Artifact Affected	System
Environment	Limited wireless connectivity, high packet loss
Effect of the Action	The handheld notifies the user about the error
Response Measure	The handheld does not crash and displays an appropriate error message

ID	16
Stakeholder Priority	Medium
Difficulty Ranking	Difficult

Quality Attribute	Portability
Source of stimulus	RTC
Actual Stimulus	RTC ports the system from Android to iOS once this Android version exists
Artifact Affected	System
Environment	Normal operation
Effect of the Action	The system is ported to iOS
Response Measure	RTC implements iOS-specific parts with changes confined to one package in the system architecture.

ID	17
Stakeholder Priority	High
Difficulty Ranking	Easy
Quality Attribute	Usability
Source of stimulus	Untrained handheld end-user
Actual Stimulus	The handheld end-user, untrained, unfamiliar with the Ptolemy tool but familiar with handheld devices, runs a demo
Artifact Affected	System
Environment	Normal operation
Effect of the Action	The demo runs without crashing
Response Measure	The end-user only needs a URL and credentials to access the server in order to run the simulation and can operate the demo with no guidance.

ID	18
Stakeholder Priority	High
Difficulty Ranking	Easy
Quality Attribute	Reliability
Source of stimulus	Ptolemy simulation error
Actual Stimulus	The handheld user is running a model that experiences a Ptolemy simulation error
Artifact Affected	System
Environment	Normal operation
Effect of the Action	The demo aborts the normal execution
Response Measure	A Ptolemy exception is thrown and a brief user-friendly message describing the exception is shown to the user on the device

ID	19
Stakeholder Priority	High
Difficulty Ranking	Challenging
Quality Attribute	Reliability
Source of stimulus	Network connection
Actual Stimulus	The handheld user is running a model
Artifact Affected	System
Environment	Abrupt network connection loss
Effect of the Action	The simulation terminates

Response Measure	Both the server and the client detect the connection loss and gracefully end the simulation on both ends

2.2.3 BUSINESS CONSTRAINTS

Description	**Difficulty Ranking**	**Comments**
Must not include GPL or similar licensed code	Easy	Ptolemy is to remain a freely available, ongoing, and open-source project and additional licenses cannot be more restrictive.
Must use Ptolemy as the simulation engine	Challenging	The Bosch RTC has been using Ptolemy for their research for a long time, as it is very similar in concept than the ASCET commercial tool, it is however open-source.
Must complete the project in the time allotted by the MSE program.	Challenging	The MSE program is 16 months, starting from 09/2010 until 12/2011. The projects must be completed within this timeframe.
Must scope the project according to available team resources.	Challenging	The MSE HandSimDroid team has a predefined time allocated to work on the project, which is roughly 4300 person-hours distributed over the 16 months

2.2.4 TECHNICAL CONSTRAINTS

Description	**Difficulty Ranking**	**Comments**
Must work within the confines of existing Ptolemy architecture	Challenging	Additions to the system may require changes to the core classes of Ptolemy, making the necessary testing more extensive and modifications more risky
Handheld application must run on Android OS	Challenging	Android applications are primarily written in Java (much like Ptolemy)
Handheld devices that are currently available have limited	Challenging	Performance (I/O), memory, battery life

resources and capabilities		

3 HIGH-LEVEL ARCHITECTURE

The system must support two major high-level functionalities:

1. Ability to run the simulation on the handheld
2. Ability to customize the UI of a model for a handheld device

Knowing that Ptolemy is hardware resource intensive, it's evident that the first requirement is the riskiest one with regards to meeting the performance quality attribute response measure. As a result, the team came up with three possible approaches to address this issue:

1. Use the code generation capabilities of the Ptolemy to generate an optimized version of the model.
2. Since Ptolemy is based on Java, port the whole Ptolemy to Android and assume that the handheld would execute it fast enough as defined in the performance quality attribute.
3. Use the client/server model to offload the complex processing to a server and use the handheld as a thin client that is only responsible for displaying output, providing input, and controlling simulation execution.

Running the whole Ptolemy on Android seems to be the simplest solution. The system would meet the extensibility quality attribute because everything would be ported to Android. Obviously, classes using java.awt and java.swing would need to be modified to support Android but besides that, the whole architecture would support all possible extensions that Ptolemy currently supports. The risk is meeting the performance quality attribute because of the limited hardware resources on currently available devices.

Code generation is a second approach that we could use which is supposed to remove the overhead of Ptolemy-specific functionality and generate code only needed for specific model simulation. The risk again is meeting the performance quality attribute since we don't know if the optimization process makes the simulation execute fast enough. Also, this approach increases the complexity since the current code generation process does not seem to support UI-specific functionality and does not support Android. We would need to add this functionality ourselves and it's not clear if the architecture of the code generation would easily allow this extension. Since generated code must be re-compiled each time to be runnable on Android, this also makes the system a lot less usable. Considering the business case that the system might be used for testing and configuration purposes of difference engines, the models are likely to change often and thus would require re-compilation following each change. This significantly impedes the usability quality attribute since the process is complicated and cumbersome. We also found that code generation was an experimental feature in Ptolemy since it

didn't reliably generate code for many models or support all Ptolemy directors. This is a risk that could impact our time constraint.

The last solution is to use the client/server approach and offload the heavy processing to the server and use the handheld as a thin client only responsible for displaying results and providing sensor input to the simulation. This approach is significantly harder than the complete port but seems to be easier than the code generation one although we still don't have any guarantees that the performance quality attribute would be met.

In order to clear up these unknowns and make the decision regarding *architectural style* of the system, the team conducted three experiments. The results of the experiments suggest that all core non-UI specific classes can be ported to the Android without any modification. This indicates that extensibility quality attribute can be promoted this way. However, the experiment showed that the speed of simulation is unacceptably slow compared to the desktop version. As a result, we decided to abandon this approach.

The next experiment was to code-generate a very simple Ptolemy model. The Ptolemy II cg code generator that we tried on a simple model failed because of lack of support for Complex data types. Therefore, this approach seemed to be too risky considering Ptolemy has had three implementations of code generators, and they didn't seem to work correctly for all models. Considering that we have a time-driven project, there is a high likelihood that we would not have enough time to fix the code generation to support all models and make the generated code portable to Android with support for custom UI designs. Moreover, it's still not clear if the code generation would help us in promoting the performance quality attribute since we don't have evidence if the generated models perform better. Consequently, this approach was abandoned.

The last approach seems to be the most viable one. It would help in promoting the performance quality attribute by removing the computational bottleneck but it's unclear if it would be enough to meet the performance requirement. With this approach we are trading the computational bottleneck on the handheld for the network throughput/latency bottleneck.

Additionally, the current Ptolemy architecture does not seem to have any support for such distributed operation. As a result, the team has to build one from scratch, which increases complexity. Since we knew that the communication between the client and server could be a bottleneck with this approach, we decided to conduct an experiment that sends large amount of data in real-time from server to client. We tested different protocols and all protocols were able to keep up with regards to the throughput, but had different latency ranging from less than a second to five seconds which was satisfactory for our client. Since the client/server approach seemed to be the most viable solution, we decided to build the system architecture around it.

Our system is built using the client-server architectural pattern, specifically using the MQTT protocol and message broker for publish-subscribe functionality.

The figure below uses a mixed perspective since tiers in our system map one-to-one to the hardware.

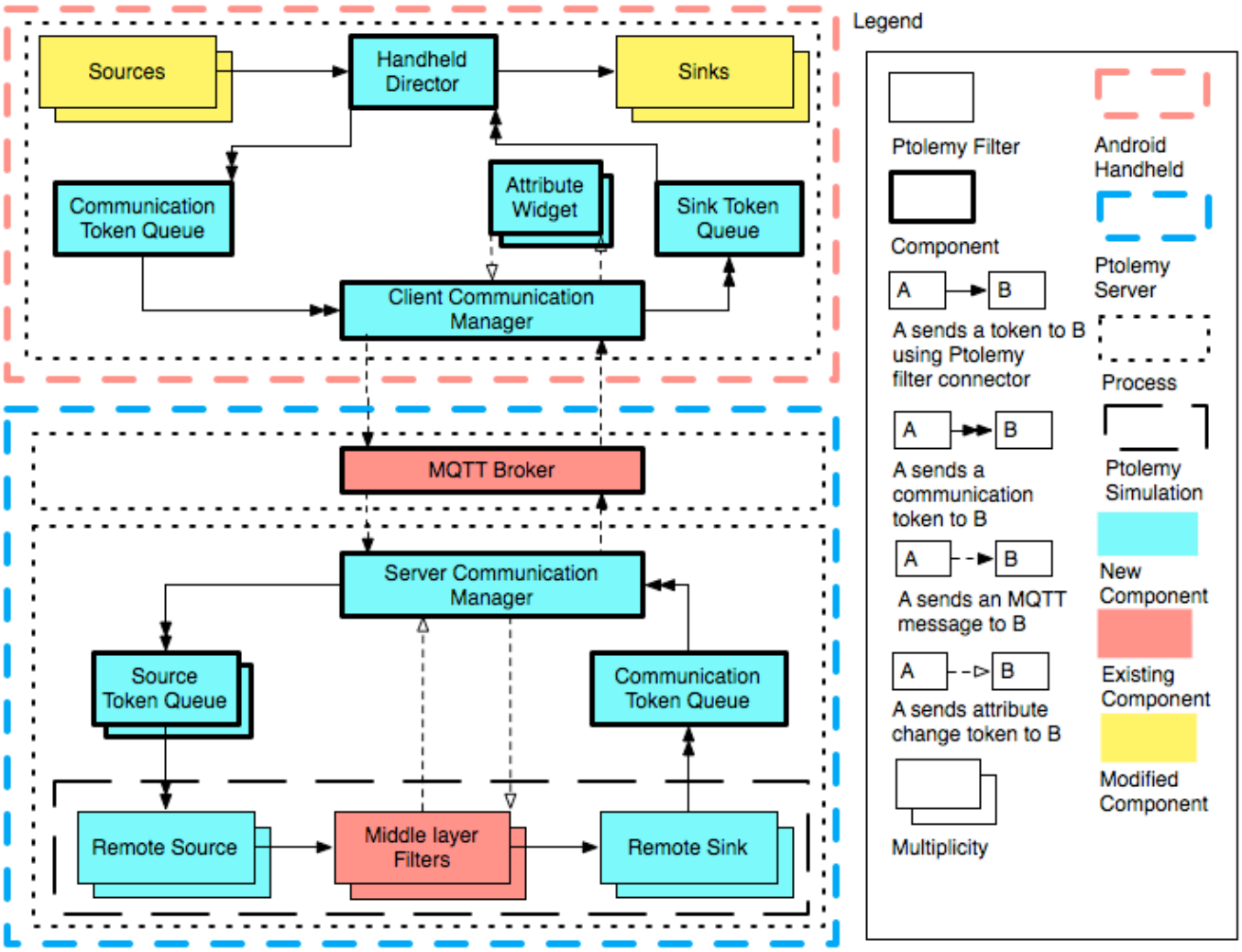


Figure 4: Dynamic perspective of HandSimDroid Ptolemy

Element	**Responsibility**
Client Communication	This component is responsible for receiving from and sending tokens to the

Manager	server. The received tokens are special communication tokens, which indicate to which actor on the handheld where they need to be routed. The responsibility of this component is to put those tokens into source or sink queues such that handheld director can read them afterwards. The component is also responsible in serializing and de-serializing the tokens to the binary format in order to send them as MQTT message. The component will send a batch of tokens each time in order to minimize the number of MQTT messages. Based on our experiment, we know that it's easy to flood the broker if small messages are sent to often. This component is also responsible for creation and initialization of the sink and source actors and remote attribute widgets based on the specification defined in the UI design file. This component is also responsible for updating attribute widgets for attributes that are present in the middle layer filters or accepting attribute change tokens based on user input into the attribute widget.
Server Communication Manager	The responsibility of the component is similar to the client communication manager with regards to serializing and de-serializing tokens and routing them to correct actors. Additional functionality of this component is to accept control commands from the handheld which could be the following: ● Start, pause, and stop the simulation ● Provide list of available models ● Load a model and accompanying UI layout ● Provide description of UI elements to be loaded on the handheld Similar to the client communication manager, it sends attribute change tokens from the Attribute Widget to the middle layer filters and back when the filter changes attribute values.
MQTT Broker	Protocol-specific component to support communication between the client and server. This component is responsible for publish/subscribe functionality. This component runs in a separate process from the server process. We depicted it as running on the same server but technically it can run on other machine with Wi-Fi connectivity.

Middle layer Filters	Ptolemy filters/actor that are not sinks or sources. They accept tokens either from a source or from other middle layer filter, process the token, and forward them to the next filter. They encompass the bulk of the simulation's computation requirement. As a result, their processing is offloaded to the server.
Remote Source	This actor will replace any source actor if that source actor is included in the UI design file. The remote source actor will accept tokens from the actor equivalent to one it replaced but running on the handheld by reading the communication tokens from its queue, unwrapping them into regular tokens, and forwarding them to the middle layer filters via the appropriate ports. The actor will run according to the model of computation defined in the loaded Ptolemy model. As a result, it can run either as a separate thread or via call-return style of computation.
Remote Sink	This actor will replace any sink actor if the sink actor is included in the UI design file. The remote sink actor will send all tokens it receives to the actor equivalent to one it replaced but running on the handheld by wrapping the token into a communication token and putting into its queue. The actor will run according to the model of computation defined in the loaded Ptolemy model. As a result, it can run either as a separate thread or via call-return style of computation
Sources	These actors are sources that read data from the environment, store that data in the token, and send them to the subsequent filter/actor.
Sinks	These actors are sinks that accept final output tokens. Usually they are in the form of graphs or similar UI elements or act as file writers.
Source/Communication Token Queue	This is a concurrent queue from which both readers and writers can work concurrently. In the case of server, a queue is created for each sink or source because we don't know at what rate they generate tokens and when they do it. As a result, a separate queue is needed for each one so they could do it at their own pace. On the handheld, the director is developed by us and we can use one queue and route tokens appropriately to/from each actor.
Handheld Director	This director is responsible for firing sources and sinks and pushing tokens

	to/from them by reading communication and sink queues.
Attribute Widget	This widget will display current value of the remote attribute of the middle layer filter (if the attribute is included in the UI design file) and also allow changes to this attribute. It would receive change tokens from the client communication manager when the attribute in the middle layer filter changes. It would also send change request to the server after user changes the widget's value.
Android handheld	Android handheld device running the simulation with a custom UI specific to certain model.
Ptolemy Server	This server (hardware) will host the Ptolemy simulation engine and the MQTT broker.

Table 1: Components' responsibility catalog of the dynamic perspective (running simulation)

Connector	**Responsibility**
A sends a token to B using Ptolemy filter connector	This connector is responsible for sending tokens from one filter to another. This connector is build-in into the Ptolemy.
A sends a communication token to B	This connector is responsible for sending communication tokens that wrap regular tokens but also contain routing information from one component to another.
A sends an MQTT message to B	This connector is responsible for sending an MQTT message, which is a binary message, to a specific topic that other side is listening to.
A sends attribute change token to B	This connector routes tokens that specify changes in attribute values.

Table 2: Connectors' responsibility catalog of the dynamic perspective (running simulation)

From the client side, user will be able to select which model and associated UI design to load. Based on that a Ptolemy model would be loaded on the server in the same way the current desktop Ptolemy does it. However, after that the server communication manager will dynamically replace all sinks and sources with remote sinks and sources respectively. Those removed sinks and sources will be created on the

client side instead. As a result, the core simulation would run in exactly the same way because those remote sinks and sources will be just regular actors from the Ptolemy's simulation engine's point of view and thus they would conform to all execution rules but in addition to that transparently send or receive tokens on the mobile side. This promotes extensibility because this way any actor or simulation type can be used on the server side as long as the remote sources and sinks could be placed instead of the local ones.

This client/server architectural pattern promotes the performance QAS from computation point of view because hard processing is done on the server but inhibits QAS by introducing a new communication bottleneck.

4 First-level Decomposition

4.1 Ptolemy Simulation

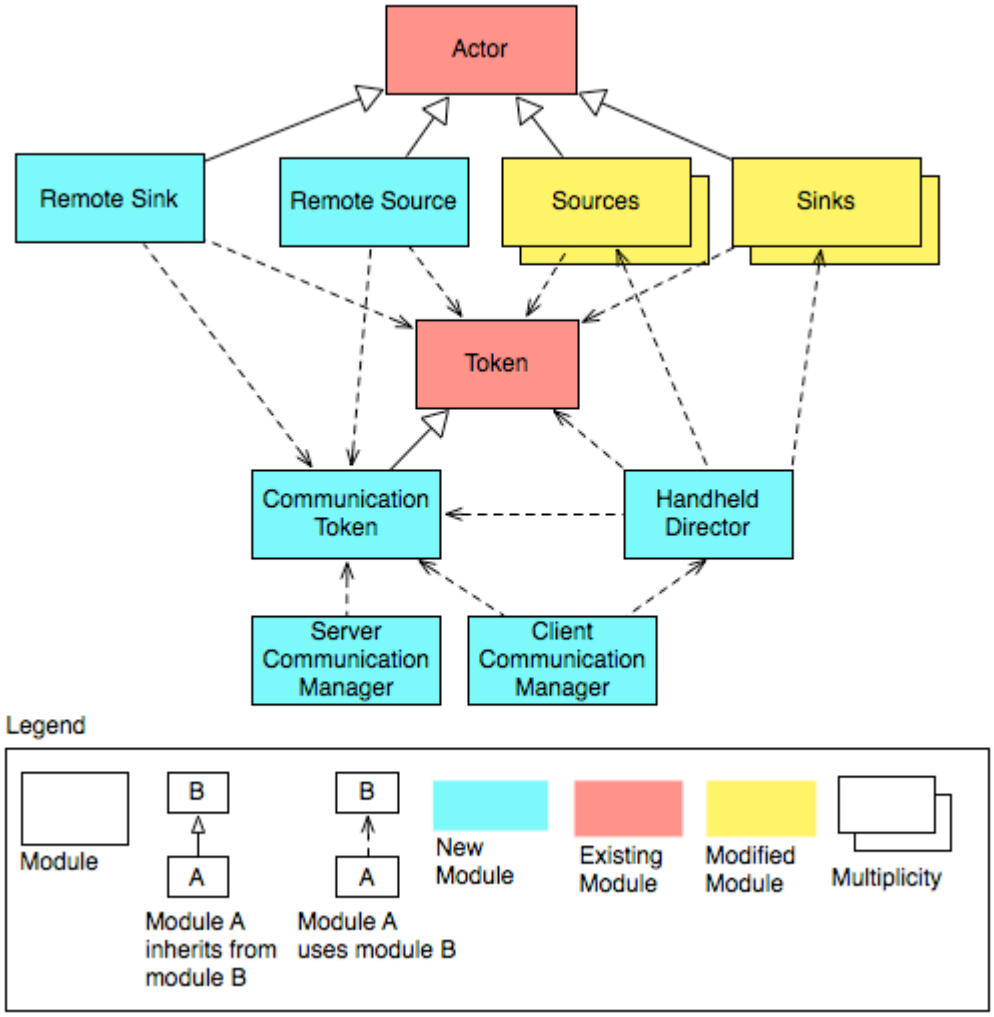


Figure 5: Classes/interfaces needed for simulation

Element	Responsibility
Remote Sink Remote Source Sources Sinks Server Communication Manager Client Communication Manager Handheld Director	Please refer to the dynamic perspective element catalog of the figure 1
Actor	Filters in Ptolemy are called actors. This is core Ptolemy class for building new filters by extending it. All actors/filters need to extend this class in order to be used in the simulation engine
Token	This is standard Ptolemy mechanism of communication between actors. Actors can communicate using different token types. Tokens are immutable - they can't be changed once initialized.
Communication Token	Remote Sink and Source actors will use the communication token to wrap original token and also include routing information along with it such as to which actor and which port this token needs to be routed on client or server side. The Remote Sources and Sinks can wrap multiple tokens into this token that if they have multiple ports and each contain a token.

Table 3: Responsibility catalog of the static perspective (running simulation)

Most components in Figure 4 map directly to classes in this diagram. We plan to add CommunicationToken which will contain routing information from where the original token came from and where it needs to go. For reliability reasons, all tokens received per iterations will be encapsulated in one token so when it's delivered on the other side an actor can safely fire without worrying that this could break the simulation. For performance reasons, communication tokens will be sent in a batch. Based on our prototype, we discovered that sending tokens one-by-one throttles the client, server, or broker.

The director on the server side orders Remote Source actor to run. The actor will read from its queue and will block until it receives its communication token from the client side. This is done in order to

preserve the invariant of the simulation and make sure the current system is working reliably. This impedes performance if tokens are not delivered fast enough. Based on our prototype, we found that the performance should be sufficient and blockage should not cause major delays. By having a separate queue for each sensor, we are decoupling ourselves from the director that could be used on the server because the system does not care in which order actors read token from their queues. Similarly, after the remote sink has received its token, it would package it as a communication token and put it into outgoing queue. The server communication manager would read from or write to these queue periodically (based on iteration from the director) and send/receive those tokens to/from the client.

On the client side, client communication manager will instantiate the needed sinks and sources and put the handheld director in between them. The handheld director would continuously call those actors and send and receive tokens to/from these actors by reading source and sink queues. It would also work with remote attribute widget for displaying certain attribute values of the middle layer filters.

4.2 COMMUNICATION PROTOCOLS

We have evaluated three protocols as possible candidates for communicating between the client and server. We decided to select MQTT because of its reliability and extensibility features with an adequate performance metric. The following sections would summarizes each protocol:

4.2.1 MQTT (OUR SELECTION)

MQ Telemetry Transport (MQTT) is a lightweight broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement. MQTT is built on top of reliable connection oriented TCP/IP network protocol. These characteristics make it ideal for use in constrained environments, for example, but not limited to:

- Where the network is expensive, has low bandwidth, or is unreliable
- When run on an embedded device with limited processor or memory resources

Features of the protocol include:

- The publish/subscribe message pattern to provide one-to-many message distribution and decoupling of applications
- A messaging transport that is agnostic to the content of the payload
- The use of TCP/IP to provide basic network connectivity and guaranteed delivery and ordering
- Three qualities of service for message delivery:
    - "At most once", where messages are delivered according to the best efforts of the underlying TCP/IP network. Message loss or duplication can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost, as the next one will be published soon after.
    - "At least once", where messages are assured to arrive but duplicates may occur.

    - "Exactly once", where message are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
- A small transport overhead (the fixed-length header is just 2 bytes), and protocol exchanges minimized to reduce network traffic
- A mechanism to notify interested parties to an abnormal disconnection of a client using the **Last Will and Testament** feature

4.2.2 XMPP

XMPP is the Extensible Messaging and Presence Protocol, a set of open technologies for instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data.

Compared with MQTT protocol, the XMPP protocol has more security and other features, and thus more advanced. The XMPP is open sourced and their open source communities have contributed rich standards and applications. The XMPP is good for real time applications, like instant messaging and real time gaming.

XMPP was originally developed in the Jabber open-source community to provide an open, secure, spam-free, and decentralized alternative to the closed instant messaging services at that time. Originally limited to character data, today XMPP is capable to encode binary data into XML as well, introducing some artificial overhead. MQTT circumvents the issues with binary data by allowing any kind of data in the payload.

4.2.3 TCP/IP

TCP/IP is another protocol that the team considered. TCP/IP (Transmission Control Protocol/Internet Protocol) is the basic communication language or protocol of the Internet. TCP/IP is a two-layer program. The higher layer, Transmission Control Protocol, manages the assembling of a message or file into smaller packets that are transmitted over the Internet and received by a TCP layer that reassembles the packets into the original message. The lower layer, Internet Protocol, handles the address part of each packet so that it gets to the right destination.

TCP/IP uses the client/server model of communication in which a computer user requests and is provided a service by another computer (a server) in the network. TCP/IP communication is primarily point-to-point, meaning each communication is from one point in the network to another point or host computer. TCP/IP is said to be "stateless" because each client request is considered a new request unrelated to any previous one (unlike ordinary phone conversations that require a dedicated connection for the call duration). Being stateless frees network paths so that everyone can use them continuously. (Note that the TCP layer itself is not stateless as far as any one message is concerned. Its connection remains in place until all packets in a message have been received.)

Since both MQTT and XMPP are both based on underlying TCP/IP connection, it's expected to deliver better performance in terms of throughput and latency since it does not have other protocols' overhead.

4.2.4 TRADE-OFF ANALYSIS

Performance vs. extensibility

According to the results from the communication experiment the throughput from using TCP/IP was better that that of MQTT. The transfer of voice signal from the desktop machine to the handheld took about 2 seconds while for MQTT the time lag was around 3-4 seconds. On the other hand MQTT provided better support for implementation in network with multiple sensors since this protocol is specifically developed for such networks. Since in the future this system is expected to be used in networks with multiple sensors. Also extensibility is of higher priority as compared to performance according to quality attribute requirements.

Security vs. performance

XMPP has better security mechanism to provide secure transfer of data across the network but such a security is at the cost performance in terms of speed of data transfer. But the quality attribute of performance has much higher priority than security so MQTT is the better choice. Please note that security can be added to MQTT later using an external to MQTT SSL protocol.

Time vs. performance vs. reliability

Since all three mentioned protocols are based on TCP/IP connection-oriented protocol, all of them promote reliability quality attribute, because the messages are guaranteed to be delivered in order.

TCP/IP would deliver the best performance but no security and takes a considerable amount of time to implement support for multiple sensors required for our project.

XMPP provides an end-to-end secure by default protocol, which suggests the slowest performance out of the three. MQTT would allow us deliver the comparable performance without adding significant security communication overhead. Time to implement MQTT is the shortest because our project already uses MQTT broker. Thus, MQTT promotes time and reliability quality attributes better than other protocols.

4.2.5 ANALYSIS OF THE SELECTION

The decision to select the protocol was based on the trade off analysis motivated by the quality attribute requirements of the system. After analyzing the results of the communication experiment with the client and presenting their relative strengths and weaknesses, MQTT was selected as the most suitable protocol. The major reason for the selection of the protocol was the extensibility that is provided by MQTT in terms of ability to collect data from different sensors not just he handheld device. MQTT is better optimized to work in networks where data is collected from multiple sensors. Such protocol that provides ease of extensibility is better suited for the vision that Bosch has for this system.

4.3 UI DESIGNER

Our main concerns for the UI designer are extensibility QAS 5-6, usability QAS 8 and time constraints.

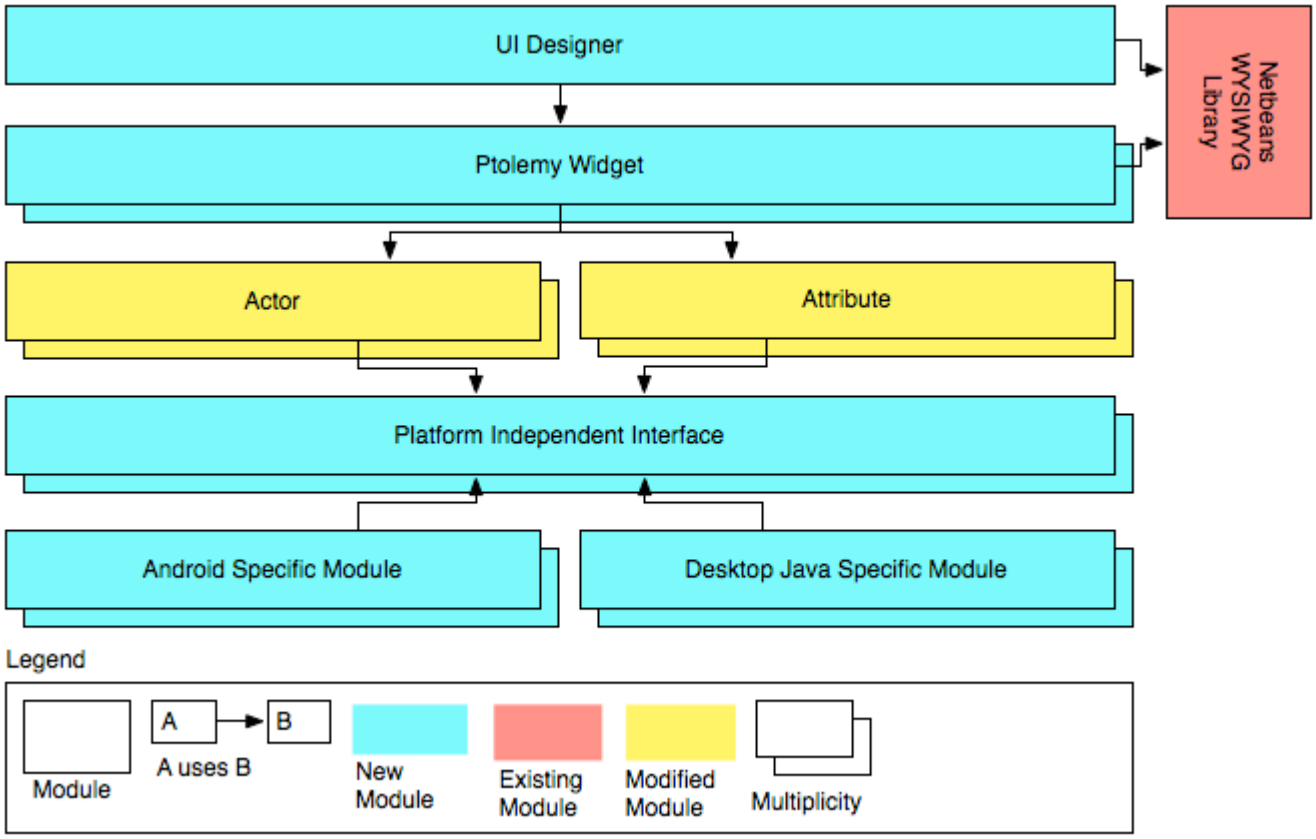


Figure 6: UI Designer from a static perspective

Element	**Responsibility**
Actor	Please refer to the table 1
Attribute	Attributes are configuration parameters that actors have. Users can change them in order to tweak behavior of an actor.
Platform Independent interface	These interfaces are a set of platform independent interfaces that separate actors from the platform dependency. Examples of platform dependency could be GUI elements, sound and etc., which are handled differently on Android and desktop Java. We will develop this intermediate layer and modify needed actors to use them instead. Implementation of these interfaces will be

	supplied dynamically when the system loads based on the platform.
Android Specific Module Desktop Java Specific Module	These are platform dependent implementations of the interfaces. Correct implementation will be injected during run time based on the platform.
Ptolemy Widget	A special widget associated with every platform independent actor or attribute will be written that will define how the widget looks on WYSIWYG UI designer layout.
UI Designer	This class will be responsible of laying out Ptolemy Widgets and allowing user to add/remove/move any them.
Netbeans WYSIWYG Library	This library provides a set of modules to quickly develop WYSIWYG UI designer.

Table 4: Responsibility catalog of the static perspective (UI Designer)

Figure 6 depicts high-level architecture of the UI designer. We plan to reuse the current architecture of Ptolemy II that defines how each actor needs to be placed within a container. This is enforced via Placeable interface. However, currently there is no support for placing individual parameters within a container and in order to fix this we plan to add an additional widget that will support this functionality.

In order to make a current actor runnable on Android, we plan to remove all platform dependencies by inserting an intermediate layer in between the actor and the platform dependent module. For the structural analysis of this problem, please refer to Appendix C. The approach can impede performance because of additional layering and increases complexity. However, it promotes extensibility such that any actor that is modified to use platform-independent classes can run on Android without modification. In the future when handhelds are more powerful this design will allow the user to run the whole Ptolemy on Android with minimal effort by just removing remote sinks, sources and communication managers and compiling the whole Ptolemy to run on Android.

The UI designer's Ptolemy widgets and their associated actors are completely separate. The mapping would be defined in some kind of configuration file. Thus, this makes it easy to add new actors or modify them without touching UI designer or vise versa, which promotes maintainability.

This design meets QAS 10 because the individual widgets don't depend on the size of the screen and only require a fixed size container to place themselves. As a result, it's possible to build UI for any

screen size. It also possible to add support for new sensors for some newer device, just by developing a new actor along with appropriate Ptolemy widget and then developing a model and UI design for it.

4.3.1 WYSIWYG DESIGNER LIBRARIES

Our main concerns for the selection are time constraints (we want to save time by using COTS component), licensing requirements of the library, and the extent to which it's maintained.

4.3.1.1 Netbeans (our selection)

Netbeans provides a set of libraries to quickly build WYSIWYG application by just including three jars. The IDE is under constant development (last commit on the day of the writing) and based on our experience with Eclipse and Netbeans and subjective opinion, Netbeans provides far superior user experience for designing UI for Java applications. Drag and drop interface is very intuitive and customizable.

The license is GPL & CDDL. Any changes made to the core library must be contributed back. However, we don't plan to make any changes but rather just use those jars to build our UI designer. As a result, we are not constrained by CDDL and only need to include the licensing notice in the package.

The Visual Library has quite good documentation and tutorials, which would minimize learning curve and help us meet the time constraint. It also seems that the architecture of the visual library allows for easy extension. Based on our tutorial we did, basic application with draggable widget could be created with 30 lines of code.

4.3.1.2 Eclipse

Eclipse has a project for building GUI called Visual Editor Project. The design intent is to provide architecture that could support any kind of GUI builders. Again based on our subjective opinion, the user experience is rather subpar compared to the Netbeans. Also it's not clear if we could use this project's library without Eclipse by building the GUI builder into Vergil directly. The project seems to be in stale state (based on the last release date), and it seems that the GUI builder tool called Window Builder released by Google would become a de-facto standard tool in Eclipse soon since the project is quite active and the user experience is much better. Currently, there is a proposal to incorporate it as an eclipse project: http://www.eclipse.org/proposals/tools.windowbuilder/

4.3.1.3 Ptolemy's built-in implementation

Ptolemy's build-in UI builder uses a library called FormLayoutMaker. The library is intended to easily create user interfaces using a tabular layout. This allows defining to which cell each components needs to go and its stretchability/layout properties. As a result, resulting layout can support variety of sizes. However, it's not a WYSIWIG tool and seems to be overkill for quickly creating user interfaces per

model. The current implementation overcomes this problem by generating a default layout for any model but this is not suitable for a mobile because of the screen size constraints. Also the project was last update in 2006 and we are not sure if the author is available in case if we have problems or if documentation is not good enough. However, the current tool also includes support for Placeable interface and querying and listing all attributes and actors, which we could reuse. Also it has support of writing the UI configuration to a file, and the approach is also useful to know.

4.3.2 *TRADE-OFF ANALYSIS*

Usability vs. complexity vs. reuse

It seems that Netbeans has the best usability. It's the most updated project and has quite good documentation and tutorials, which would minimize learning curve. We know that it can be included as a separate library and used within Vergil. The complexity is low based on our tutorial we did. However, Netbeans might impact our schedule because we have to rewrite most of the stuff currently available in the Ptolemy's UI designer. However, we could minimize that by reusing code from it since the general notion and approach would be the same.

Eclipse's VEP project is stale, and it does not seem to be possible to include it as a stand-alone library since it depends highly on Eclipse.

The current implementation of the UI tool in Ptolemy does not meet QAS 10 because tabular layout cannot be mapped directly to the way the design would look on a handheld.

As a result, Netbeans is a best option for meeting QAS 10, 12 because of good WYSIWYG layout tool and ability to easily customize layout for different screen sizes.

4.3.3 *FILE FORMAT*

Currently Ptolemy stores the simulation model within an xml file. The file is processed using the module called MoML. It's an extensible system that allows addition of new parsing capabilities for new nodes and attributes in the xml files. The module supports both serialization and deserialization of model data to/from an xml file. Furthermore, the current version of UI designer also uses the same capabilities to store its UI configuration within the same model file.

Consequently, in order to promote maintainability QAS 12, it makes the most sense to reuse the same capabilities for the new UI designer. The data can be stored within the file along with the model file.

All visualizable actors will need to have additional property nodes that defines the following attributes needed for our UI designer and UI loader:

- X coordinate
- Y coordinate

- Width
- Height
- Z-order

The top-level actor must have the following attributes:

- Screen orientation
- Screen width
- Screen height
- Screen density

All actor attributes must have all attributes that actors have and in addition to the following attributes:

- Enabled/Disabled
- Required/Not Required
- Listen to change or not

Taken as a whole, the MoML system provided by Ptolemy is quite extensible in a sense that other applications that don't know how to process certain nodes just ignore them and it's relatively easy to add support of new nodes.

5 Analysis of the System

Extensibility (QAS 5-6) vs. Performance (QAS 7) vs. Time and Complexity vs. Extensibility (QAS 11) vs. QAS 9

By putting a layer between platform dependent classes and actors, we are promoting extensibility since any actor that uses those platform-independent classes (since we are going to develop reusable library of this components such buttons, labels and etc.) could immediately benefit from this design. Those actors should be immediately usable on the handheld and UI designer after recompilation.

This could potentially inhibit performance because of the layers that method calls need to go through. Also due to this layering and constraint to support both Android and Desktop Java at the same time, the code could not be optimized to support faster mechanisms of one platform not available on the other.

The complexity is increased because we have to invest time in developing the platform independent layer, which might be hard because of the major differences between UI architecture of Android and desktop Java. This impacts our schedule, which is fixed.

According to QAS 11, if some new feature of Android 3.0 comes, it might not be usable immediately because a similar functionality is not available on the desktop and as a result this functionality could not be exposed via platform independent layer. Similar scenario is valid on the desktop Java too. As a result, if there is need to make this functionality available, the only viable solution is to develop a platform dependent actor for the platform.

QAS 9 is promoted because of this layering. If Android 3.0 comes out and has new functionality, this functionality is either available immediately because platform dependent classes use it or can be exposed via the platform independent layer. The important part is that even if code breaks because Android changes its API, the code breakage will be contained to the platform dependent classes and will be easy to fix.

However, we believe this design is a good choice for the following reasons:

- Performance impact is negligible (based on our experiment)
- Extensibility QAS 5-6 have higher priority than QAS 11
- We don't need to port all actors and as a result, only could develop the platform independent classes for the actors needed. Our design allows to incrementally adding support to the remaining actors as needed.
- Since we are putting Ptolemy actors on the Android, we are allowing future extension to allow the whole Ptolemy on the handheld in the future once hardware is capable to support this

Reliability (QAS 4, 13, 18, 19) vs. Performance (QAS 7)

By using MQTT protocol, we promote reliability because MQTT has functionality called Last Will and Testament which is a special message that the broker sends out when one of the clients disconnect. As a result, even if the Ptolemy server or client crashes, the other side would receive a message about that and be able to end the simulation gracefully (QAS 19). If server crashes, the handheld application would be able to receive user-friendly message (QAS 13). This also helps in meeting time constraints because complexity is reduced since the functionality is already built into the protocol. The protocol can inhibit performance however based on our experiments the response measure (throughput and latency) were within acceptable range.

The protocol also handles connectivity errors and provides an easy way to recover such as the MQTT Broker would queue up MQTT messages for a certain period if client is able to reconnect. This would help in meeting QAS 13.

Communication Managers would handle communication of tokens and control messages between client and server. As a result, in case of a runtime exception in Ptolemy simulation engine, a control message could be sent between the two indicating the description of the problem (QAS 4).

Time vs. Usability (QAS 8) vs. Extensibility (QAS 5-6)

Currently Ptolemy has an experimental UI designer. We could save time and just use that and add missing functionality; however, from the usability point of view, it does not meet the QAS 8 such that there is no one-to-one mapping from the UI design to the actual running simulation. Also, this tool was not designed with Android in mind and thus does not meet extensibility quality attributes. As a result, in order to meet QAS 8, we need to use Netbeans UI libraries that provides WYSIWYG support but in order to save time we could reuse portion of this tool's architecture and design.

Portability (QAS 16) vs. Extensibility (QAS 5-6)

We are not promoting this quality attribute scenario directly. However, because we are using MQTT communication protocol and keeping major logic on the server, it's possible to develop an iPhone client and communicate with the server using MQTT since the protocol is vendor neutral. In order to support portability QAS better (i.e. by using a thin web based client), we have to sacrifice extensibility QAS which has higher priority. As a result, we find the current design satisfactory, and if there is a need to port to iPhone the whole client package would need to rewritten.

Maintainability (QAS 12) vs. Extensibility (QAS 11) vs. Extensibility QAS (7-8)

Maintainability is promoted because we are keeping the current architecture of the Ptolemy, specifically architecture of the way Actors visualize and place themselves within container. As a result, anybody who is familiar with the architecture will be able to maintain the handheld application and UI layout designer. However, this inhibits QAS 11 because it would be harder to add new functionality available in the Android 3.0 in a cross platform way due to the requirement to develop platform independent interface. However, since QAS 7-8 have higher priority and there is always a way to add platform specific actors, we find this compromise justified.

Usability QAS 17

This QAS is promoted because with UI designer it would be possible to develop a custom UI for per model. As a result, for an untrained user, it would be possible to create a simpler UI with limited and simple controls.

6 APPENDIX A: EXPERIMENTS

6.1 CLIENT/SERVER COMMUNICATIONS

6.1.1 OBJECTIVE

This prototype has the following objectives:

- Test if it's possible to "hijack" an actor - replace an actor with a special actor that does client server communication during runtime by traversing actor hierarchy and replacing a selected actor only. Side effect of this prototype is that we get hands on experience with innards of the Ptolemy and learn and evaluate its architecture
- Measure performance in terms of latency and throughput of communicating real time data from the server to the handheld android. We decided to capture audio from the microphone and send it to over the network to be played back on the other side.
- Develop small Android application that can act as a sensor
- Evaluate different protocols and compare their speed using this architectural approach

6.1.2 APPROACH

We divided the prototype into four parts and assigned each part to each team member. The prototype was divided in the following way:

- Develop a small application that "hijacks" a simple actor and send its tokens to the handheld using server side communication interface
- Develop a server side communication interface
- Develop a client side (Android) communication interface
- Develop an Android client application that sends audio captured from the microphone over the network and also playbacks audio data received from the server

Two team members working on the communication protocols each selected XMPP and MQTT protocols for evaluation and based on their analysis, they would develop communication interfaces for a selected protocol

6.1.3 ANALYSIS

- Is it feasible to "hijack" an actor?
- What's benefit of selecting one protocol over another in terms of latency, throughput, features, reliability, and extension points?

Results

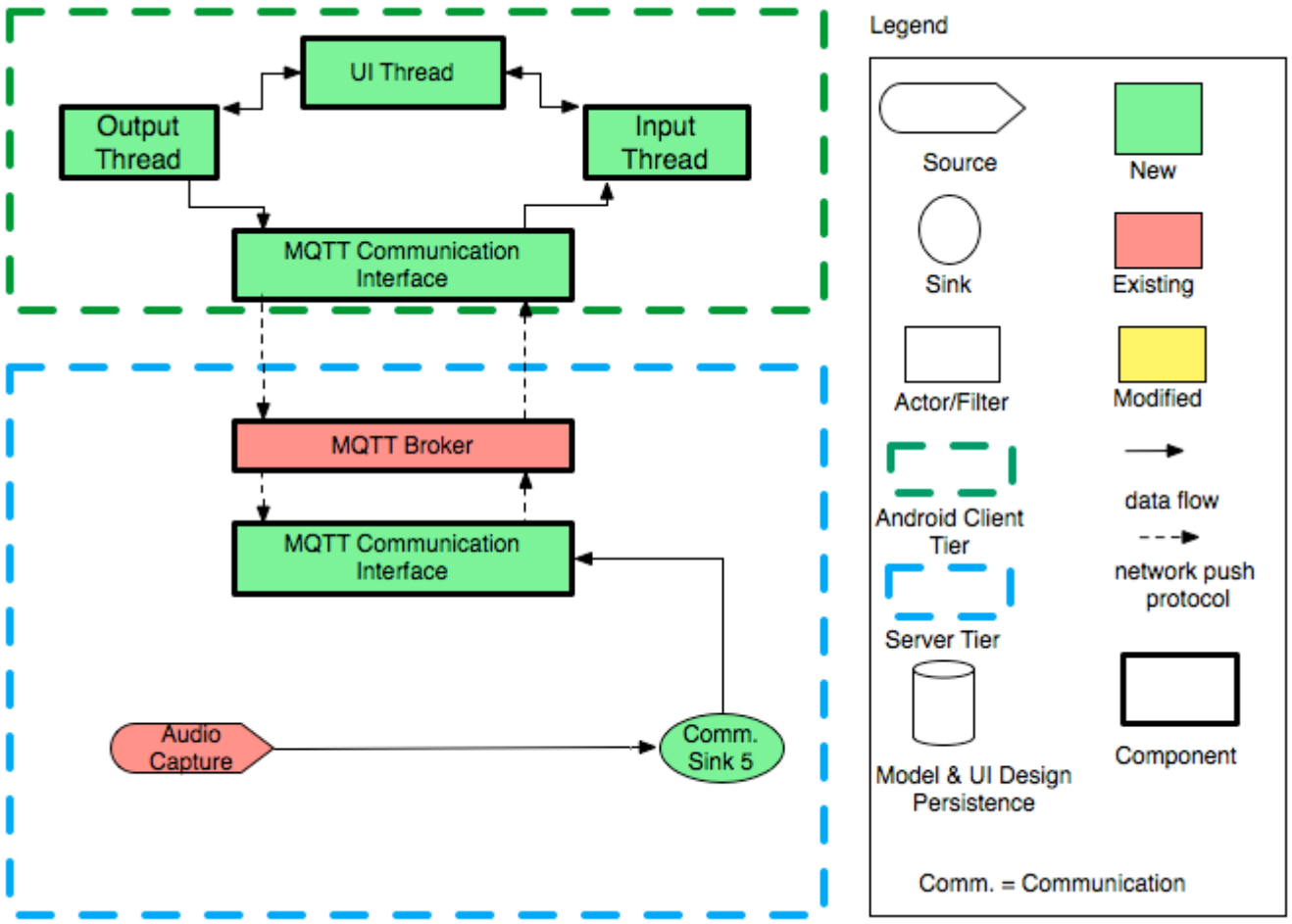


Figure 7: Client-server communication experiment

We ended up developing a prototype that has the architecture similar to the dynamic perspective diagram depicted above. We selected MQTT protocol based on the trade-off analysis.

Initially, very simply Ptolemy model is loaded consisting of an audio capture actor and a sample sink connected to it. After that, the sink is replicated dynamically with communication sink that pushes its tokens in a batch via MQTT communication interface to the handheld by first converting them to a binary stream.

The MQTT broker handles communication between its clients, which are Ptolemy server, and Android handheld.

On the handheld side, the input thread would receive those tokens, convert them back to the original format and play them. The output thread captures sound from the microphone and sends them via the communication interface. At the moment of writing, the Ptolemy server was not listening for this data.

We were able to make the following conclusions regarding the communication protocol and the architectural approach:

- It's possible to replace dynamically actors by traversing actor graph. However, further investigation must be done in terms of how to transform different tokens, how to correctly initialize different kinds of relations and ports (IO, Typed port/relations). In addition, we need to figure out how to read tokens in a batch instead of one by one. Some reference to transfer size was found that specifies how many tokens must be received before firing but we were not able find how to implement this functionality in the communication sink. Right now instead, 500 tokens are buffered before sending them as a batch.

- MQTT protocol seemed to work reasonable well and was able to play audio on the handheld captured on the server with 1-4 second lag. However, we didn't investigate how the protocol behaves when there are multiple streams of data being received and send at the same time.

- It's clear that advantage of MQTT protocol over XMPP is that it communicates using binary data compared to an XML ASCII that XMPP has to use this. ASCII increases size of data that needs to be communicated by 30% because of the need to convert to base64. This must be done for almost all data types. For example, double data cannot be represented precisely as a string. As a result it's necessary to first convert it into binary and then output the binary in the base64 format. All this can be avoided if the data is send with binary format which MQTT uses. Disadvantage of the binary format is that we need to develop a protocol that defines how to communicate with each other using binary commands and how to format each message. With XMPP it's easier since formatting is already defined with XML. Since we have to implement all this functionality ourselves, MQTT becomes almost as complicated as TCP/IP. The only winning point of MQTT over TCP/IP is that it allows communicating with external sensors easily. However, it's unclear how useful this future extension is to our clients compared to possible speed advantage of the TCP/IP and no licensing and third party library requirement. Although we have not tested TCP/IP, we believe that it will be at least as fast as MQTT since it's based on TCP/IP too. UDP is not considered because we need to guarantee correct ordering and delivery of all tokens.

6.2 PTOLEMY ON ANDROID

6.2.1 OBJECTIVE

The prototype has following objective:

- Compile and run Display actor infrastructure on android
- Want to see if actor can be moved with little modification and eventually want to move whole simulation to android
- Check the timestamp by printing with received token
- Measure performance in comparison to desktop equivalent
- Measure performance overhead that will result from Ptolemy actor
- Identify dependencies on GUI for actor
- Identify whether director stub is necessary on android
- Use Mosquito broker and compare its performance with RSMB

6.2.2 APPROACH

- Run audio capture actor on desktop and then transfer the tokens to the Display actor running on the Android over the network using MQTT protocol.
- Create a communication interfaces on either end of the network. The communication interface on the server will convert the tokens to binary data do that it can be transferred over the network using the MQTT protocol. The communication interface on Android will convert binary data from the MQTT subscribe to tokens so that the Display actor can use them.
- Modify the Display actor to remove the dependencies on Java swing and replace it with Android specific GUI implementation.

6.2.3 ANALYSIS

- With the current architecture is it possible to meet the quality attribute for performance?
- Does to simulation run more than 10% slower on the handheld with the current architecture?
- To what extend is the display actor dependent on the Swing GUI?
- Does the Display actor need a director to run the simulation?

Results

We were able successfully able to port the Ptolemy actors on the android by removing the java.awt dependencies in the actor classes. In fact we went a step ahead and used sound data captured from the microphone of the desktop and transferred it over the network through the communication interfaces to the Ptolemy actors running on the handheld. [If you got this far, you are invited to eat pizza with us; please email us]. The reason we used the Android sound actor instead of display actor was that we wanted to compare its performance with the previous experiment where we used sound actor on the desktop. The audio capture actor on the handheld read the sound data in the form of tokens and converted it into sound output. Following are the findings from the experiment:

- We found out that we need a director on the handheld to control the flow of tokens to the sink. The director is responsible for firing sources and sinks and pushing tokens to/from them by reading source and sink queues. We were able to create a simple director for the experiment. We found out that it is fairly easy to implement a director on the handheld.

- Another major finding from the experiment was that the latency between recording the sound to a source actor on the desktop to producing the sound output on from the sink actor on the android was almost similar to the previous prototype when there were no Ptolemy actors on the handheld. The latency was still around 3-4 seconds. So there was no significant effect of putting the Ptolemy actors on the handheld.

7 APPENDIX B: SECOND-LEVEL DECOMPOSITION

This appendix describes the second-level decomposition of the system that, while it's important to develop the understand the end product in more detail, requires deeper knowledge of the Ptolemy and HandSimDroid architecture, and is not necessary for understanding the project's overall architecture.

The descriptions and decisions made here are not final, under construction, and were used to identify possible unknowns and lower-level solutions for

7.1 APPLICATION LEVEL COMMUNICATION PROTOCOL

Throughout the application we use three different kinds of communication channels:

- **Communicating simulation data**. These are potentially the source actors defined in the used Ptolemy model, the sensor data gathered from the handheld, and the data pushed to the sink for visualization.
- **Control commands**. These are the simulation controlling commands between the handheld and the server, such as starting, pausing, stopping, or simulating one cycle commands.
- **Remote attribute configuration**. Since modification of attributes is done on the handheld, but the simulation itself is on the server, these changes need to be communicated.

These channels are sometimes connected, or example remote attribute configuration commands might need to trigger control commands at the same time.

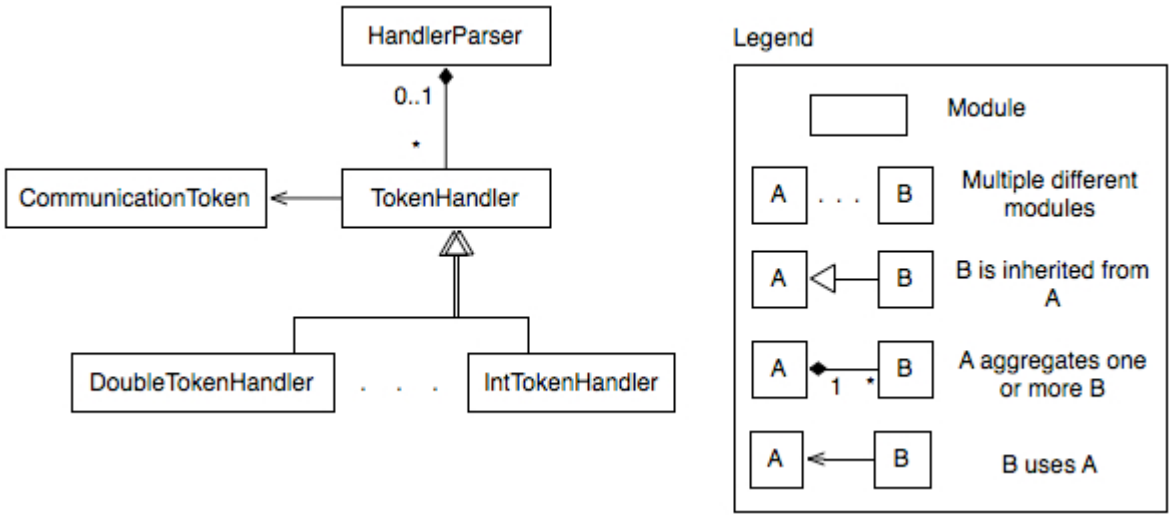
7.1.1 COMMUNICATION SIMULATION DATA

Ptolemy uses tokens to facilitate data communication between actors. Token is the base class for data capsules and they are immutable. There are many different kinds of tokens, but they can be grouped the following way:

- **ScalarToken**: These tokens can be associated with a unit, and can be converted losslessly to another type.
- **MatrixToken**: Useful for creating matrices of ScalarTokens.

- **AbstractNotConvertible**: These tokens cannot be converted to other token types losslessly, an example of this is the ArrayToken.
- **AbstractConvertibleToken**: These tokens can be converted losslessly to other types, but they are not associated with a unit, for example the BooleanToken.

To transfer these tokens through the chosen communication protocol, a binary parsing/unparsing method is



used.

Figure 8: Static perspective of the token parsing/unparsing modules.

Element	**Responsibility**
HandlerParser	• This is a class in the system • It contains the mapping between identification numbers and TokenHandlers in the system • It provides methods to add handlers, and to get back a handler from an identification number • It provides a method to parse and unparse the mapping to and from binary data, so it can be sent out
TokenHandler	• This is a class in the system • It provides both the signature and the commonalities of parsing tokens into bytes, and vice-versa
DoubleTokenHandler	• This is a class in the system, inherited from TokenHandler • It provides the specific parsing and unparsing implementation for double type tokens.

IntTokenHandler	• This is a class in the system, inherited from TokenHandler • It provides the specific parsing and unparsing implementation for integer type tokens.
CommunicationToken	• This is a class in the system • It contains the mapping between identification numbers and the client destination • It provides methods to add and retrieve client destination identification numbers

Table 5: Responsibility catalogue for the static perspective of the token parsing/unparsing modules.

While a handler needs to be created for each kind of token used in data transfer, adding and removing handlers are straightforward.

From a dynamic perspective, a stream is used to continuously read and write data on the Android side. As binary tokens come in on the stream, they are converted to tokens and pushed to the queue(s) for processing. For more on this see...

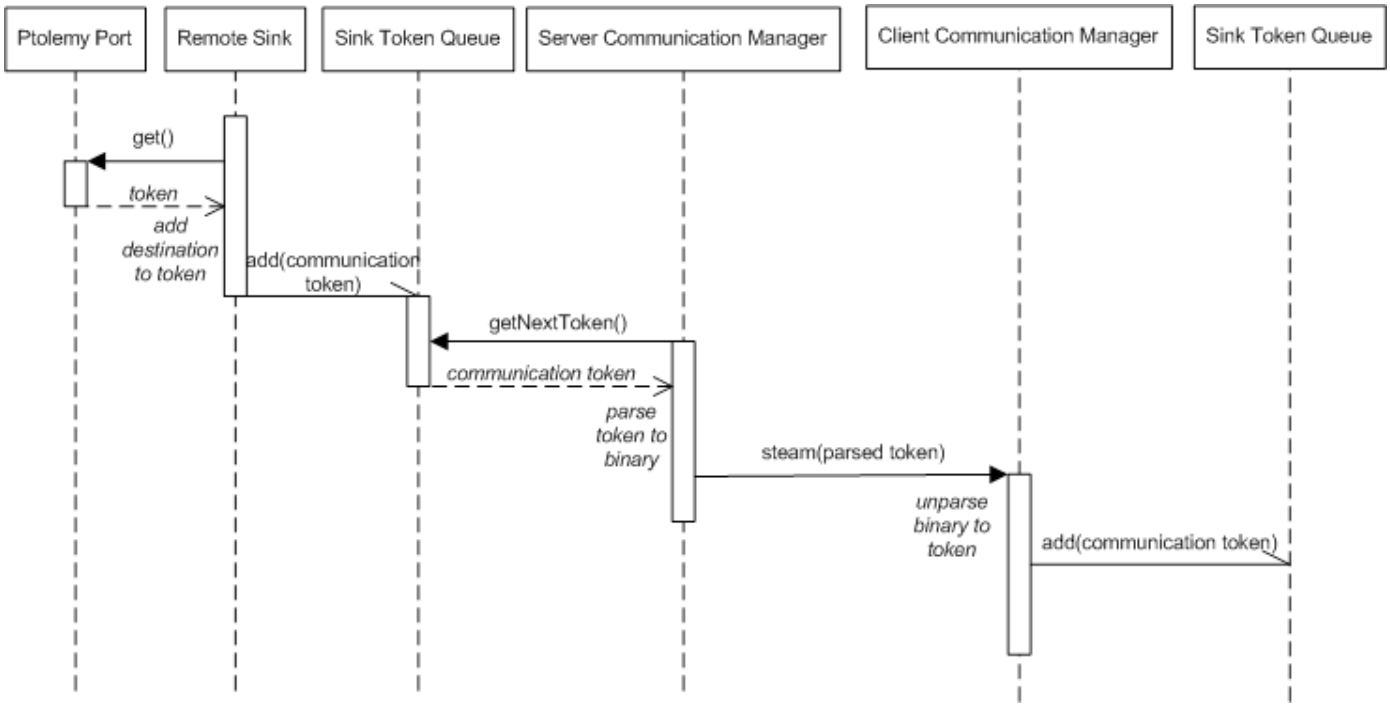


Figure 9: Rerouting token in an input port of an actor hijacked and moved to the client during simulation.

To send these through the network the following formatting strategies are *evaluated*:

- Binary
    - Parsing tokens into a binary string that contains the token type definition among with its value(s), and unparsing at the other end of the communication channel
- Serialization/deserialization
    - Serializing tokens as objects and de-serializing on the other end of the communication channel.
    - Currently we're unsure whether Android supports this.

7.1.2 CONTROL MESSAGES

The following control messages have to be communicated from the handheld to the server:

- Start simulation
- Stop simulation
- Pause simulation
- Step one in the simulation

Ptolemy's highest level of control for the execution of the model is done in the Manager class. To access this remotely from the client, a control mechanism is needed on the server-side. It's also important that these messages are communicated somewhat independently from the simulation data to ensure timely reaction.

For this command tokens are used that map directly to one of the few control commands that can be used for
simulations.

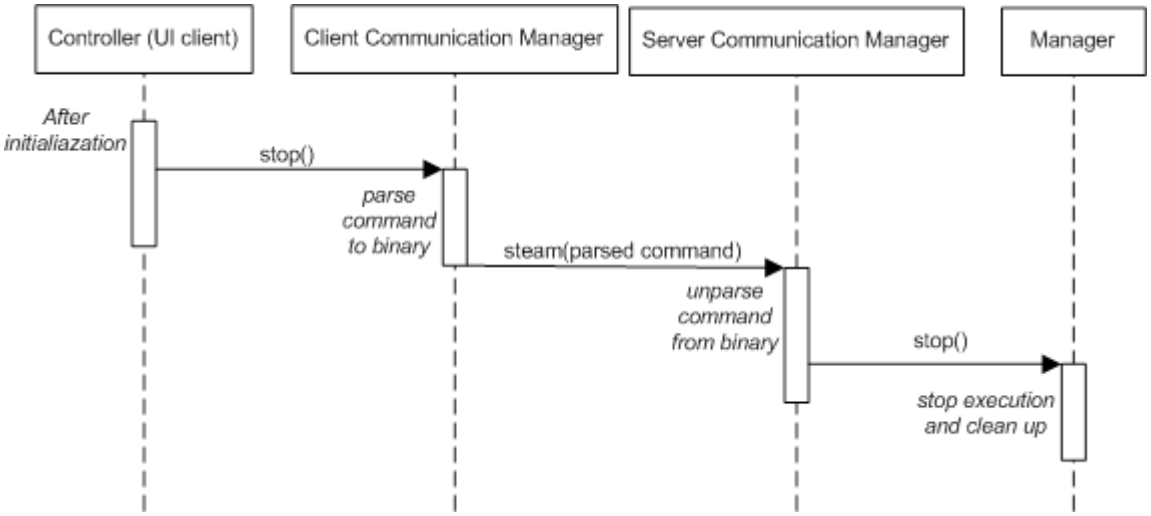


Figure 10: Example command flow during simulation

Options to consider and evaluate include:

- Asynchronous vs. synchronous command message delivery
- Use of the chosen communication protocol vs. direct TCP/UDP communication

7.1.3 Remote Attribute Configuration

While the attribute changes are in reality tokens that are sent similarly than the simulation data, it has to be handled differently both on the client and the server side.

- Use of change tokens
    - What it needs to modify
    - How
    - Formatting: see communicating simulation data
- Strategies
    - Communicate with simulation data (on-the-fly changes)
    - Communicate separately (direct one-way calls, can also be on-the-fly)
    - Pausing the simulation (or stopping) and sending changes by themselves

7.1.4 Overall considerations

- Exception handling and timeout considerations (for example stopping the simulation if control messages cannot be sent anymore between the client and the server, or if the simulation data cannot be pushed from the server to the client)

7.1.5 Trade-off Analysis

- Serialization/De-serialization algorithms
- Performance impact
- Complexity/extensibility of the algorithm

7.2 Simulation on the Handheld

Because models developed for use with Ptolemy can be computationally complex and, when active, can require a more significant amount of processing power and memory than are available on the current offering of Android-powered handsets, we have decided to offload the simulation portion to a server. The visualization of simulation data, however, must be displayed on the handheld device in near real-time and experiments to accomplish this have left us with three available approaches.

7.2.1 Summary of available approaches

- Widgets on Android (no dependency with Ptolemy)

    - Rather than using the existing suite of Ptolemy display actors, we can create our own Android-specific classes that replicate the functionality of their desktop equivalents using the available Android widgets and controls.
- Port Ptolemy's actors to Android using Abstract Factory pattern
- Port Ptolemy's actors to Android without Abstract Factory pattern
    - Load equivalent Android actor based on one in the desktop version

7.2.2 TRADE-OFF ANALYSIS

1. **Android-Specific Widgets** - Taking the approach of developing Android equivalents using the available widgets and controls has the potential for making development of the equivalent actors easier, though there is no guarantee. However, it has drawbacks that violate our quality attributes for extensibility, reliability, and portability. In terms of extensibility, we have been tasked with creating a framework that will simplify the creation of new display actors long after the project has concluded. By implementing a handful of platform-specific methods, we would ideally be able to replicate the functionality of the simulation on the desktop. Were we to take this approach and construct a suite of actors based on the currently available set based almost entirely on the tools in the Android SDK, development of new actors would need do the same. From a reliability standpoint, the resulting solution would be brittle and potentially break with future releases of the operating system. Though it is claimed that Android is backwards compatible, we have seen features moved, removed, and changed even throughout the life of this project. There is no guarantee that a widget relied upon today will behave the same way in the future. Lastly, the client, Bosch RTC, has asked that our solution facilitate porting the same functionality to other platforms. Implementing the Android functionality without following a pattern would mean that other ports effectively need to be written from scratch.
2. **Port Existing Display Actors to OpenGL** - The existing displays of Ptolemy (ex. Plotter) use a combination of Swing and AWT to display simulation outputs on the screen for consumption. Neither of these graphical toolkits is supported on Android, though a subset of the OpenGL functionality does. Without having done any experimentation into the limits of that functionality, it's hard to say whether or not it is possible to port each existing display to OpenGL ES (embedded systems) or to what degree changes would need to be made. From a development standpoint, this would increase the effort in modifying existing Ptolemy actors and for new ones, would not facilitate the extensibility and ease of development defined in our quality attribute scenarios. However, should devices continue to increase in hardware capabilities, shifting the entire simulation to the Android device would require very minimal changes. This, however, is based on the assumption that OpenGL does not drastically differ from OpenGL ES.
3. **Abstract Factory Pattern for Existing Actors** - By implementing an abstract factory pattern on existing display actors and leaving the platform-specific implementation details open, we have provided extensibility in the sense that support for new platforms can be added without modification to the underlying simulation mechanisms, particularly the directors, that manage

token passing between actors and schedule actors to fire. Instead, a developer interested in porting Ptolemy's simulation engine and display to another platform would only need to follow the provided abstract factory and create the concrete, optimized implementation that conforms to the platform's resources and available APIs. However, the pattern has its drawbacks. The pattern would need to conform to the lowest common denominator or, in this case, the capabilities and limitations of the least flexible platform. Each implementation, as well, would need to conform to the specific inputs and outputs of the strategy. This can be somewhat restrictive depending on how the strategy is defined and force implementations to follow a pattern that otherwise would not make sense given the environment.

7.3 Actor Portability

We will make actor platform independent using Abstract Factor design pattern and Google Guice for helping develop those
factories.

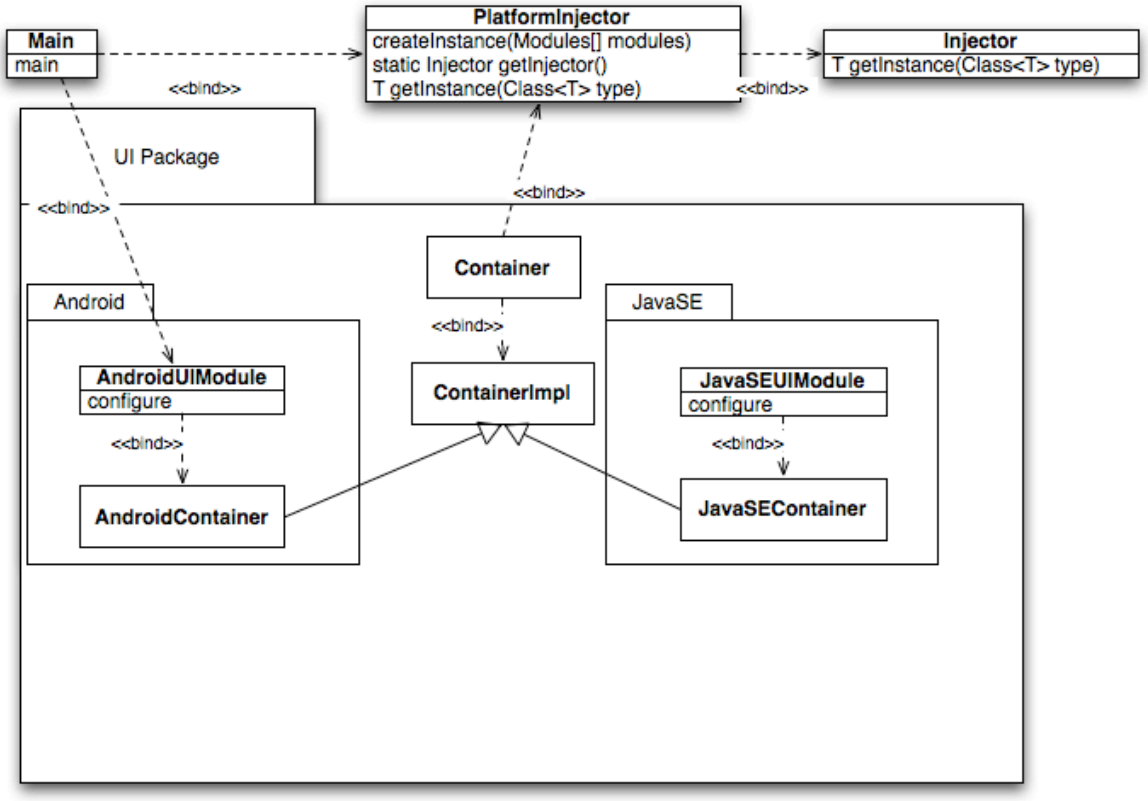


Figure 11: UML Class Diagram

Element	Responsibility
Main	Starts the system. A different one will be developed for Android that will load only Android modules using PlatformInjector

PlatformInjector	This class allows to dynamically creating a correct instance of a concrete implementation based on the bindings that were contained within the modules loaded into it. This class is singleton and initialized when system starts by loading modules into it and creating the injector
Injector	This class does actual instantiation from interfaces to concrete instances
AndroidUIModule (example) JavaSEUIModule (example)	These modules contain binding from abstract interfaces such ButtonImpl, ContainerImpl to a concrete implementation, which is platform dependent.
AndroidUIContainer (example) JavaSEUIContainer (example)	Example of platform dependent implementation
ContainerImpl (example)	Abstract interface for the container. Each concrete instance has to implement it.
Container (example)	Platform independent container anybody can use without worrying about platform dependencies
Android package	Package that will be deployed in Android Ptolemy
JavaSE package	Package that will be deployed in Android Ptolemy
UIPackage	A package that will contain a library of portable UI components

Table 6: Responsibilities of the static perspective

The above diagram is a classic example of Abstract Factory design pattern but with addition to dependency injection framework Google Guice that makes factories more modular and according to some sources faster. The architecture does not require using this framework, and we can either write our own or if Ptolemy supports something similar, use it. The license is Apache, and it seems that Google actively maintains it. Thus, this library is within our constrains.

The main idea is to have an easy way to separate Android and standard java dependent classes but at the same time reuse as much code as possible and not break other actors that we don't port. The actors that are portable would use a set of platform independent GUI classes (button, component, container, frame, etc.) or other classes (live sound for example) that we could develop initially, and that could be expanded in the future. This promotes extensibility (QAS 5-6) such that if new actors are added that support something we didn't anticipate or originally port (for example, video stream), authors would need to just define the interface and concrete implementations and add bindings to the modules. This also makes this class available for other actors to use. Also, this design will help us scope our project correctly because we don't have to port all actors now and could do this for only a portion and let others incrementally add support for Android.

When an application is compiled for a target platform, Android or JavaSE, in a single or few classes (Guice modules), one could specify bindings from interfaces to concrete implementations. For example, in the attached diagram, in the main, Android Module is loaded into injector, which specifies how to bind ContainerImpl to AndroidContainerImpl (this is just an example). All classes that want to be platform independent such as Container will use this injector to get concrete implementation of the container. This way we would have a single injector that controls which platform classes to load instead of having this scattered in multiple places. This makes it easy to maintain this library, expand later and compile to Android by not including JavaSE packages and specifying Android modules for an injector. As a result, we promote maintainability quality attribute such that it's easy to expand this library incrementally from one entry point.

8 APPENDIX C: STRUCTURAL ANALYSIS OF PTOLEMY

This analysis is used to discover the possible dependency overlaps in the current Ptolemy architecture that might make the creation of a platform independent layer unfeasible. For this analysis a tool called Lattix is used that depicts the dependencies in a matrix, and helps in describing the architecture and its modifications. This is an extract from the DSM report the team created, see References section for the full report.

8.1 ORIGINAL MATRIX

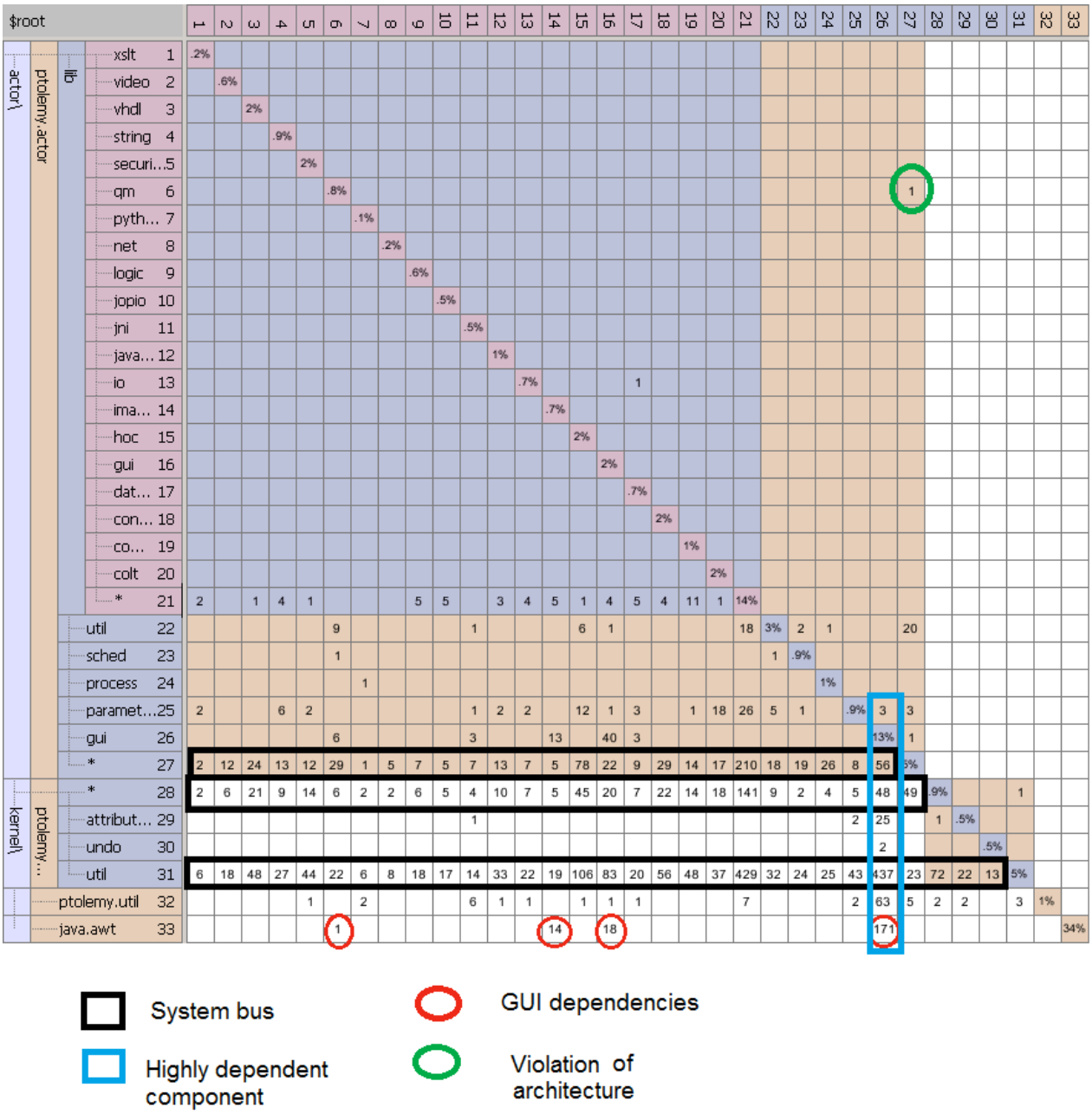


Figure 12: Original Dependency Matrix

Overall, the packages that we're focusing on have a layered architecture. Some cyclic dependencies are found, but only within the same cluster. There are three buses in the system that handle most of the connections between the modules.

The default partitioning as the packages are loaded up in Lattix show some non-cyclic dependencies above the diagonal, at first glance breaking the layered architecture. For example, **ptolemy.actor.lib.data** is dependent on **ptolemy.actor.lib.io**, but this is only because of the ordering of the sub-packages in Lattix.

There are three clusters, **ptolemy.actor.lib**, **ptolemy.actor**, and **ptolemy.kernel**. These are responsible for the main functionalities of Ptolemy we're focusing on, namely:

ptolemy.kernel adds the topology for the pipe-and-filter system used for modeling

ptolemy.actor that defines the design of filters

ptolemy.actor.lib that defines actors to be used in different domains or with different functionality

8.2 CONCEPTUAL STATIC VIEW OF THE ARCHITECTURE

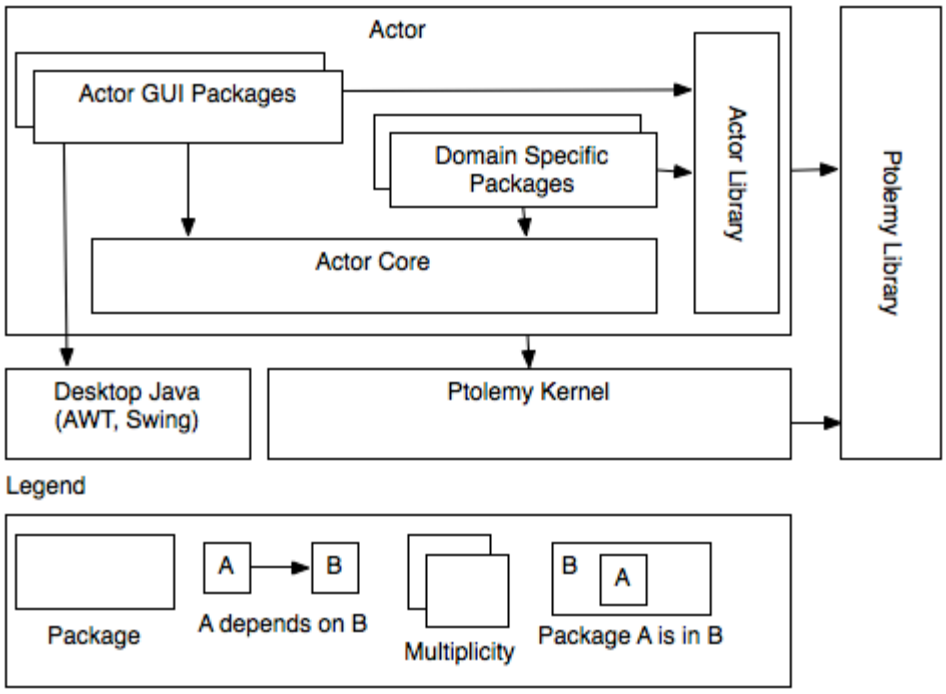


Figure 13: Conceptual Model of the Original Architecture

From the static perspective, Ptolemy kernel acts as a core of the system. The system is essentially a framework for creating different modeling systems. The kernel defines the core of the framework. Ptolemy library package provides some helper classes that are generally useful and don't belong to any particular domain.

The actor package encapsulates the core business logic of the application. It contains numerous other packages that implement simulation capabilities for different domains and also packages for visualizing the input and output of the simulation. These GUI packages directly depend on platform specific GUI libraries such as java.swing or java.awt. Although Java is created as a portable language, its GUI was not created to be used on a mobile. As a result, those packages are not available on Android, and thus we can't port Actor GUI packages to Android. However, at the same time, they contain a valid business logic which we don't want to remove or rewrite. From the maintainability and extensibility perspective, it would make sense to preserve this functionality and somehow make them portable with a minimal

cost. By looking at the dependency strength calculated by the Lattix, this approach is not very expensive from the scheduling point of view for the following two reasons:

- There are just two packages that depend on platform specific functionality
- No other package depends on the GUI packages

As a result, the changes we would have to make in order to compile them on Android would be minimal and worth the effort in order to promote maintainability and extensibility quality attributes, which have high priority in our system. Also, this helps us meet our time constrain because we don't have to rewrite separate Android GUI actors but rather make the current ones portable.

8.3 MODIFIED MATRIX

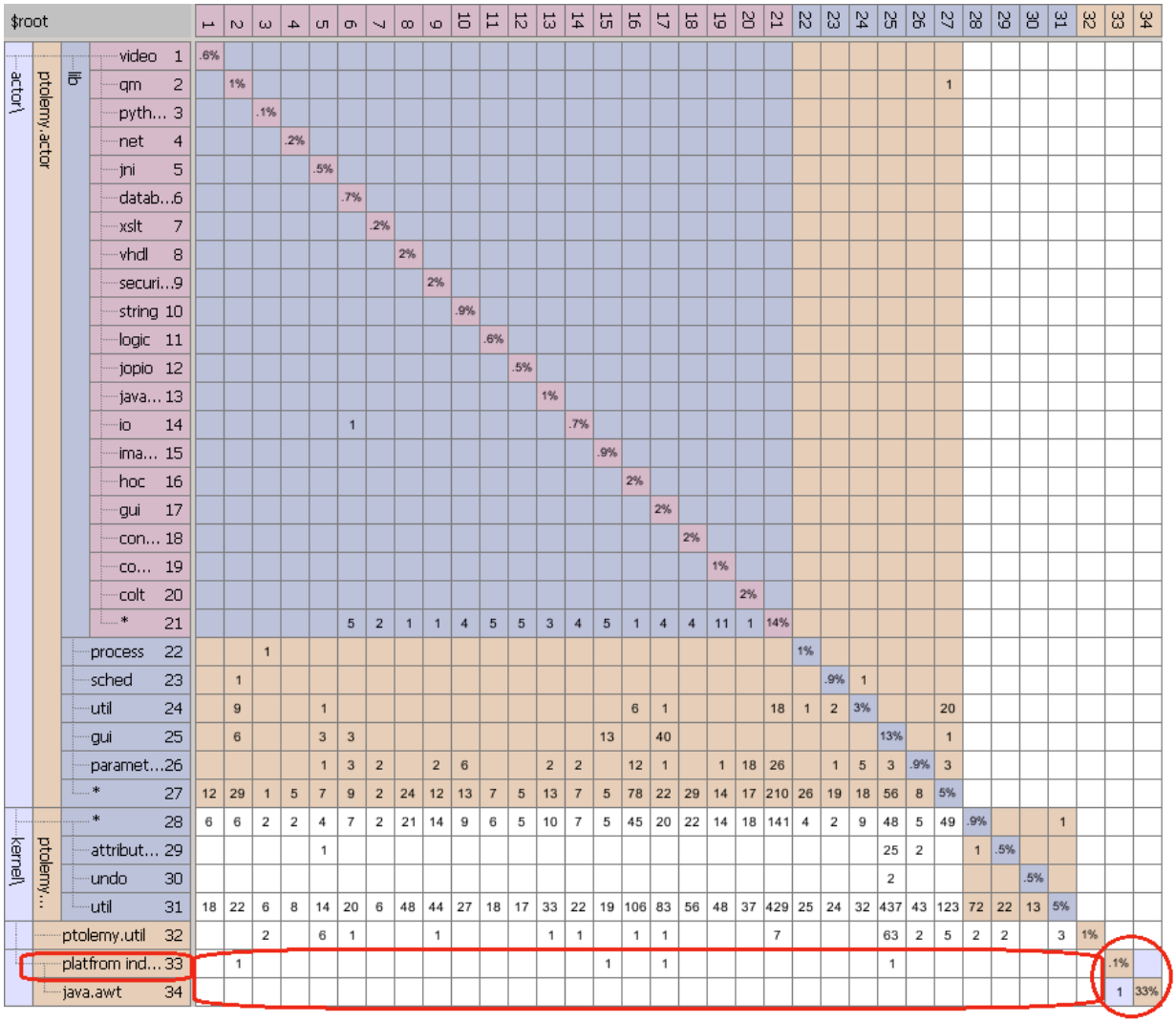


Figure 14: Modified Dependency Matrix

The new platform independent component takes the dependencies from the java.awt component, while creating a dependency between those two, creating a new cluster in the architecture. As we move on to add different platform dependent components, we will only need to create and maintain dependencies between the platform independent package and the platform dependent packages (such as java.awt or the Android SDK). The other dependencies were not modified.

Another solution would have been to actually remove those dependencies by going through each package, removing, rewriting, or rerouting those dependencies. Due to the number of packages and our time constraints, this is not deemed feasible to do.

8.4 CONCEPTUAL STATIC VIEW OF THE MODIFIED ARCHITECTURE

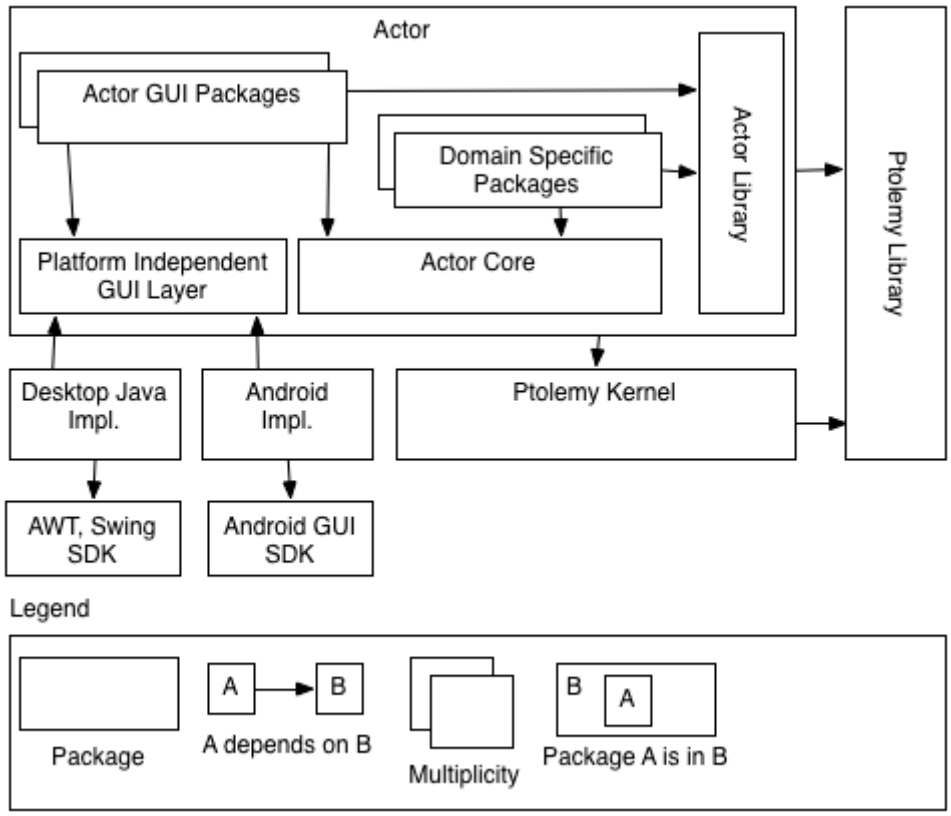


Figure 15: Conceptual Model of the Modified System

In order to port the GUI packages on Android, we decided to introduce Abstract Factory design pattern. Essentially we are adding a platform independent GUI layer between Actor GUI packages and AWT, Swing SDK. This way we could deploy different implementations depending on a platform such as Android implementation for Android application and standard desktop java for the desktop version.

Also, because we have time constraint, we can selectively port only certain GUI actors while allowing further extensions that could be developed within the appropriate platform packages. We have

calculated clustered and propagation costs and it seems that this approach would be better or as good as from propagation and clustered costs perspective.

9 GLOSSARY

Term	Description
Architecture Centric Design Method (ACDM)	Architecture Centric Design Method is a process to create system architecture iteratively, focusing on unknowns and risks within the system.
Actor	Generally speaking an executable entity. Actor can be a single entity or a composite entity consisting of different actors.
Android	Android is a popular platform used on many handheld devices such as smart phones and tabloids
ASCET	Proprietary tool that Bosch uses, analogous to Ptolemy
CDDL	Common Development and Distribution License is a free software license. Files licensed under the CDDL can be combined with files licensed under other licenses, whether open source or proprietary
Communication Token	Wraps the token with the information about which source/sink the token is destined to.
Director	Governs the execution of a composite entity
ETAS	A group that provides comprehensive and integrated tools and tool solutions for the development and service of automotive ECUs
Google Guice	A light-weight dependency injection framework for Java 5
GUI	Graphical User Interface
GPL	GNU General Public License which is intended to guarantee developer's freedom to share and change all versions of a program -- to make sure it remains free software for all its users
Manager	Governs the overall execution of a model
Model	Herein means a model of computation
MQTT	Publish/subscribe communication protocol used for message delivery

MSE	The Master of Software Engineering (MSE) is a 16 months long academic program at Carnegie Mellon University (CMU)
Placeable	An interface, it's a contract actor's UI representation in the GUI container
Ptolemy	A project that studies modeling, simulation, and design of concurrent, real-time, embedded systems with the focus on assembly of concurrent components
Quality Attribute Scenario (QAS)	Quality attribute scenario is a six-part description of a possible scenario in the system, focused on providing clear and active language, environmental detail, and specific measurements that van be used to verify the architecture and the final product.
Quality Attribute Workshop (QAW)	A technique to capture systemic properties identified in the business context in a language that is understood by the primary stakeholders of the project.
Bosch Research and Technology Center (RTC)	A subsidiary of Robert Bosch GmbH in Pittsburgh focusing on research possibilities with strong ties to the Bosch Research Center in Palo Alto, Ca.
Token	An encapsulation of a data message. Token serves to identify the message type, size and data boundaries.
User interface (UI)	Part of the software intensive system that the user interacts with, potentially causing changes in the system and receiving feedback from it.
Vergil	A graphical user interface in Ptolemy II

10 REFERENCES

1. Public HandSimDroid artifacts (Software Requirement Specification, DSM Analysis Report, Detailed Functional Requirements), http://msesrv4a-vm.mse.cs.cmu.edu/wiki/index.php?title=Artifacts

2. Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao and Haiyang Zheng, *Heterogeneous Concurrent Modeling and Design in Java (Volume 2: Ptolemy II Software Architecture)*, 2008,http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-29.html

3. Shuvra S. Bhattacharyya, Elaine Cheong, John Davis II, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, Yuhong Xiong, Yang Zhao, Haiyang Zheng, *Heterogeneous Concurrent Modeling and Design in Java*
4. *(Volume 3: Ptolemy II Domains)*, 2003, http://ptolemy.eecs.berkeley.edu/papers/03/ptIIDesignDomains/

5. MQ Telemetry Transport (MQTT), http://mqtt.org/

6. Extensible Messaging and Presence Protocol (XMPP) - http://xmpp.org/

7. Anthony J. Lattanze, Architecting Software Intensive Systems: A Practitioners Guide, 2008, ISBN-13: 978-1420045697