GAME ARCHITECTURE

GAME CONCEPT

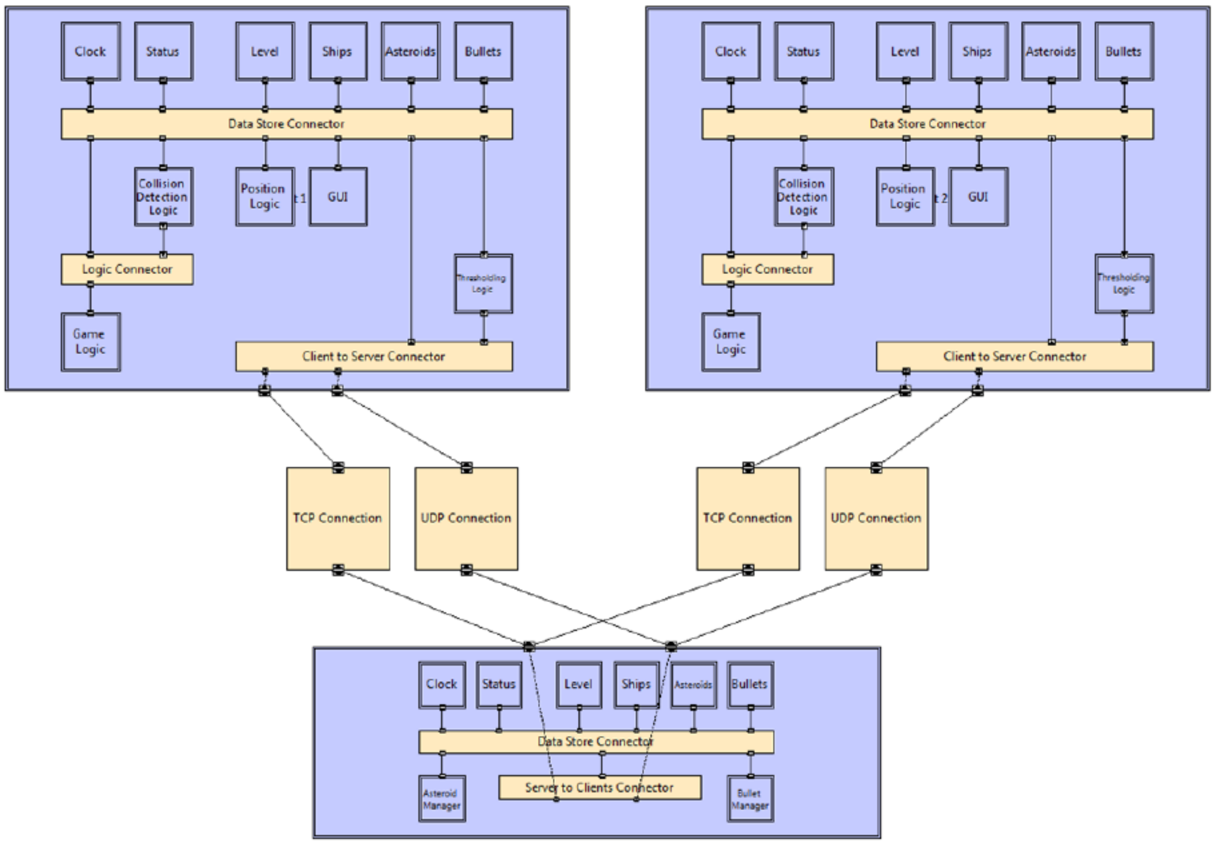
The presented game is a simple multi-player variant of the 2D game Lunar Lander[1]. Each player controls a separate ship. As additional obstacle, the players have to escape asteroids that fly over the moon surface. Their ships are also equipped with firearms, which can be used to destroy asteroids. Instead of fighting against each other, the players cooperate towards the goal of one of them safely landing on the moon surface. This could e.g. be accomplished by other players watching out for asteroids and clearing the way while one player tries to land. Players can shoot each other, but if one player's ship is destroyed, the remaining ones continue the game without that one, either until one ship successfully landed, or until all ships were destroyed.

MODELING LANGUAGES AND TOOLS

For the architecture description of the Lunar Lander game I chose the following three modeling languages / tools:

- xADL – to model the static game structure in a coarse-grained fashion, using Archstudio 4[2].
- ACME – for a finer-grained model of the networking connector structures, using ACMEStudio[3].
- UML – for behavioral models of the game initialization and GUI functionality, using Visual Paradigm[4].

COARSE-GRAINED STRUCTURAL GAME MODEL: XADL



The game is divided into a client and a server application. The above diagram shows two of clients connected to a server, all of them with their internal component structure. The game architecture does not pose any explicit limits on the number of clients simultaneously connected to the game server.

The connection between a client and a server is maintained over two channels / connections: a TCP connection and a UDP connection. While the UDP channel is used on the one hand to announce server availability (via broadcasts by the server) and on the other hand to exchange less important information (for example, ship movement updates which do not require a retransmission in the event of packet loss, because they will become obsolete with the next update), the TCP connection is established to monitor the client-server connection status and exchange important information (for example, ship or asteroid destructions and status changes).

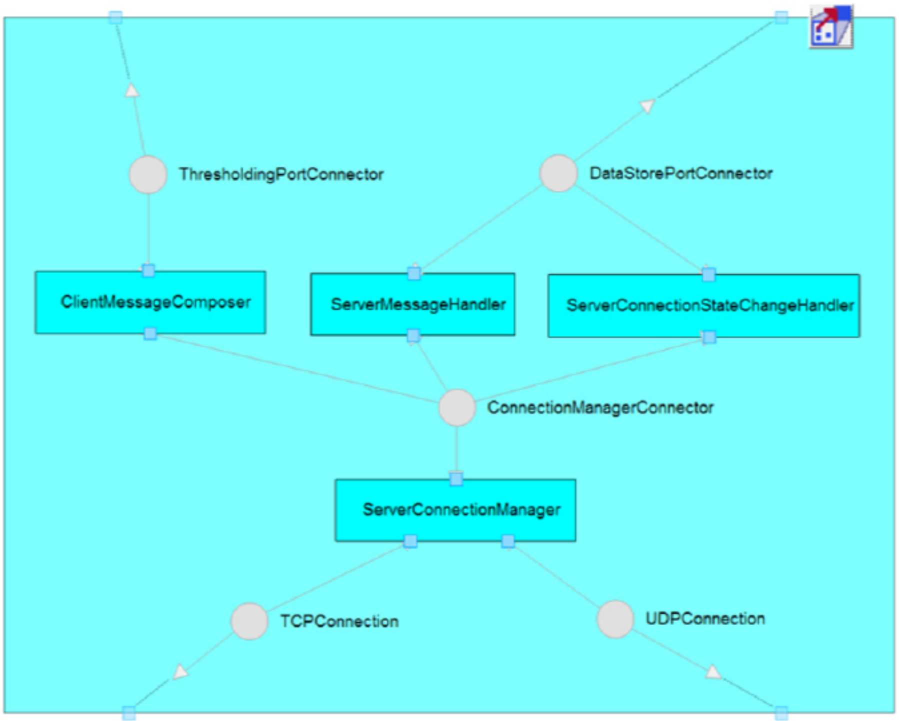
Both the client and the server implementation are based on the C2 architectural style[5] on the displayed granularity level, although the implementation of the single subcomponents (for example, GUI or Game Logic) does not necessarily conform to this style as well.

The uppermost level in both the client and server architecture consists of data store components (Status, Level, Ships, Asteroids, Bullets) and the game clock component. All of which are connected to the same connector (Data Store Connector). The data store components can process data store update requests from the components below them in the architecture, and will send data store update notifications downwards once they performed an update. The movement and position of moveable items of the game (ships, asteroids and bullets) is stored by a base position and speed vector for a certain time and an acceleration vector, to decrease the necessary network traffic for position / movement changes.

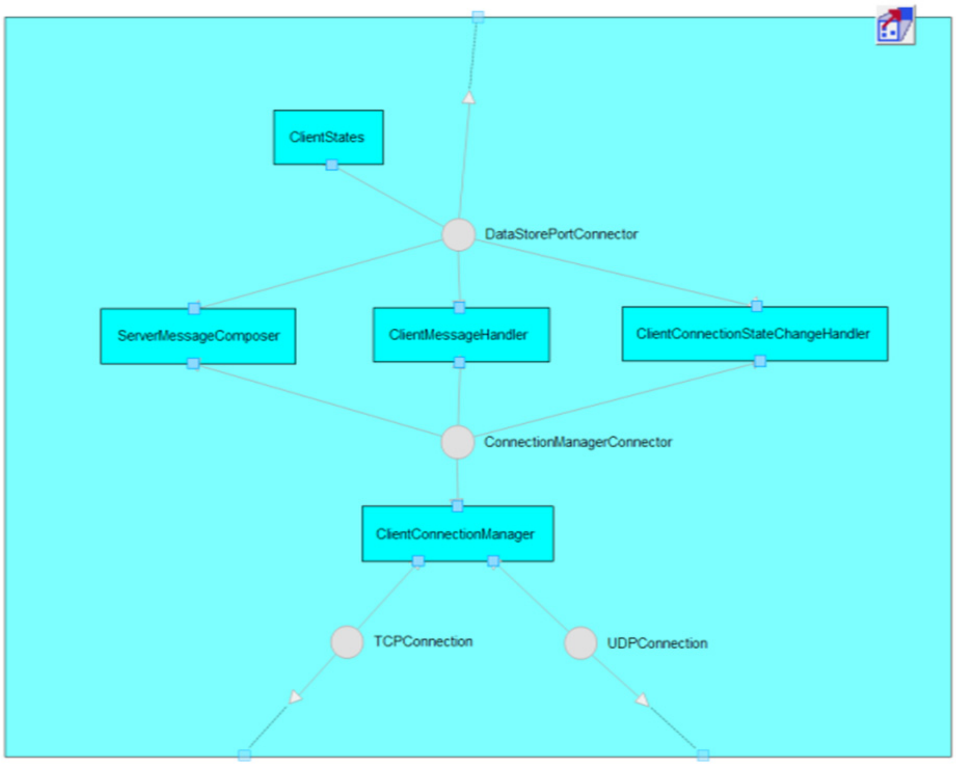
In the server architecture, the Data Store Connector connects the data store directly to the connector that handles the client-server communication (Server to Clients Connector), as well as to the components that manage Asteroids (creation and destruction on out-of-world movement) and Bullets (destruction). Other than that, the server mainly functions as a data store only, forwarding requests and notifications between the clients and managing the client connections and their data consistency. (For some more details see the game initialization model further below).

In the client architecture, the Data Store Connector connects to the Collision Logic, Position Logic and GUI components to the bottom, as well as to the Logic Connector and the Client to Server Connector, whereby update notifications to the Client to Server connector are first processed by a Thresholding component before they are passed on to the connector. The Position Logic calculates the current position and speed of ships, asteroids and bullets, by extrapolating from the last position of the items and their last speed and current acceleration vectors and writes the results back into the data store (by sending the respective update requests). The GUI component paints the user interface based on the data store state and sends update requests to it based on the user input (for example, changes in the ship acceleration or new bullets). The Thresholding component is responsible for only passing on data store changes to the Client to Server Connector when the change is bigger than a given threshold for the specific type of the change (for example, only pass on ship movement changes if the movement deviates significantly from the predicted path from the last sent position, speed and acceleration vectors). Collisions between Bullets, Asteroids or Ships will be detected by the Collision Detection component and passed on to the Logic Connector, which in turn passes on notifications form the data store and Collision Detection to the Game Logic component. This component will react upon possible collisions or state changes and evaluate their effect, issuing possible data store update requests.

STRUCTURAL MODELS OF THE NETWORK CONNECTORS: ACME

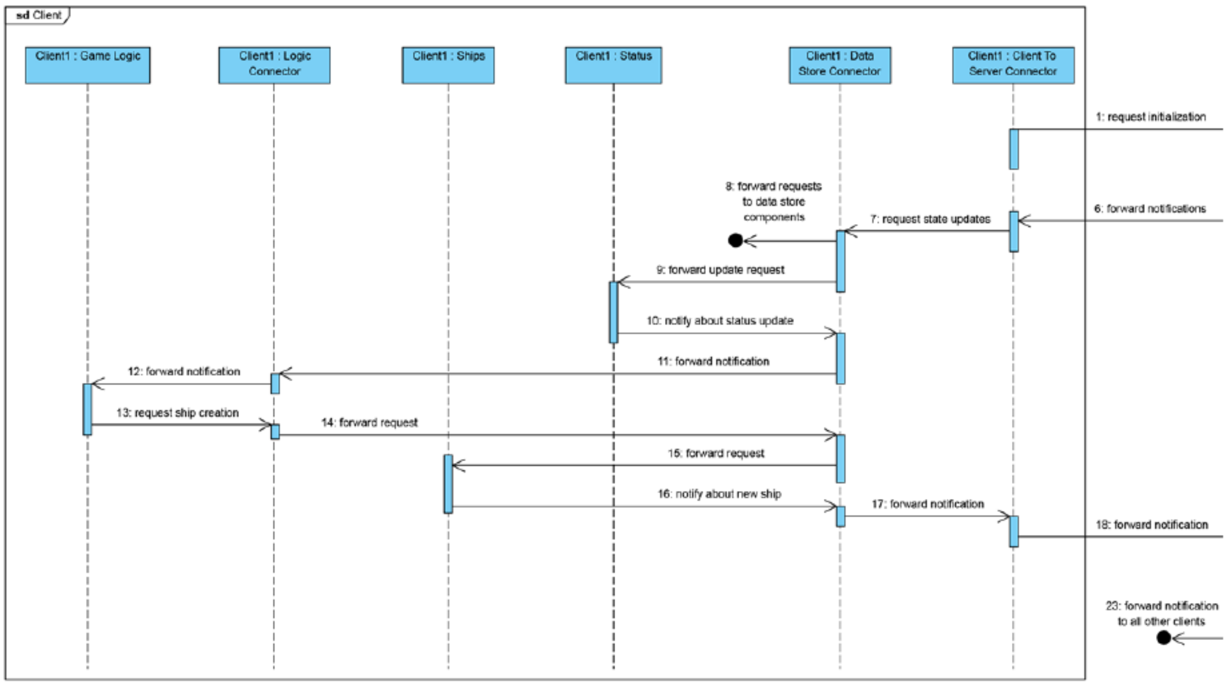


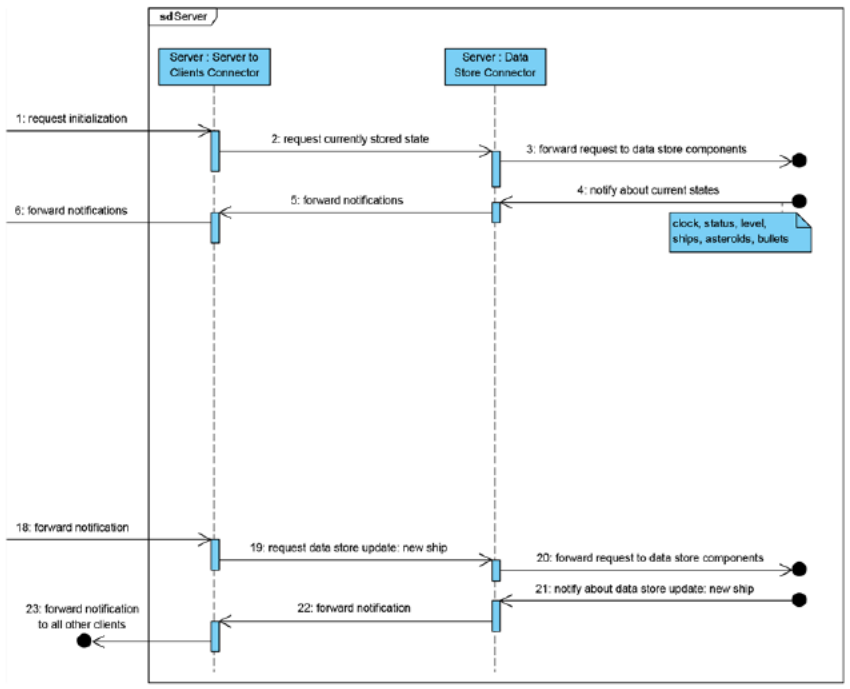
The above diagram shows the internal structure of the Client to Server Connector present in the client architecture as shown in the xADL model. Its architecture also follows the C2 style. The Server Connection Manager is responsible for serializing and deserializing messages, and transmitting / receiving them over the TCP and UDP connections to / from the server. In the case of arriving messages, it sends a handle message request to the Server Connection Manager Connector, which will forward it to the other three components. The Server Message Handler will then analyze the message and issue data store update requests to the Data Store Port Connector, which forwards them to the Data Store Connector. If the connection status changes, the Server Connection Manager requests the handling of the state change, which will be served by the Server Connection State Change Handler. For example, if the Server Connection Manager signals a connection loss, the state change handler will request a game status update respectively. Contrary, if any data store update notifications arrive at the ThresholdingPortConnector, then those will be handled by the Client Message Composer and the Server Connection Manager Connector will be notified of a new message ready for transmission (which the Server Connection Manager then sends).



The above diagram shows the detailed architecture of the Server to Clients Connector present in the Server architecture in the xADL model. It is analog to the Client to Server Connector architecture depicted above, but includes an additional Client States component, which stores the current connection states to the clients. These are updated by the Client Connection State Change Handler and the Server Message Composer can use the notifications about their change (sent by the Client States component) to broadcast the state change to all other clients. The Client Connection Handler has similar responsibilities as the Server Connection Handler in the Client to Server Connector architecture, but handles more than one connection simultaneously and provides a TCP connection / socket that the clients can connect to. The Client Message Handler evaluates the incoming messages and requests data store updates. The update notifications will be handled by the Server Message Composer and broadcasted to all other clients.

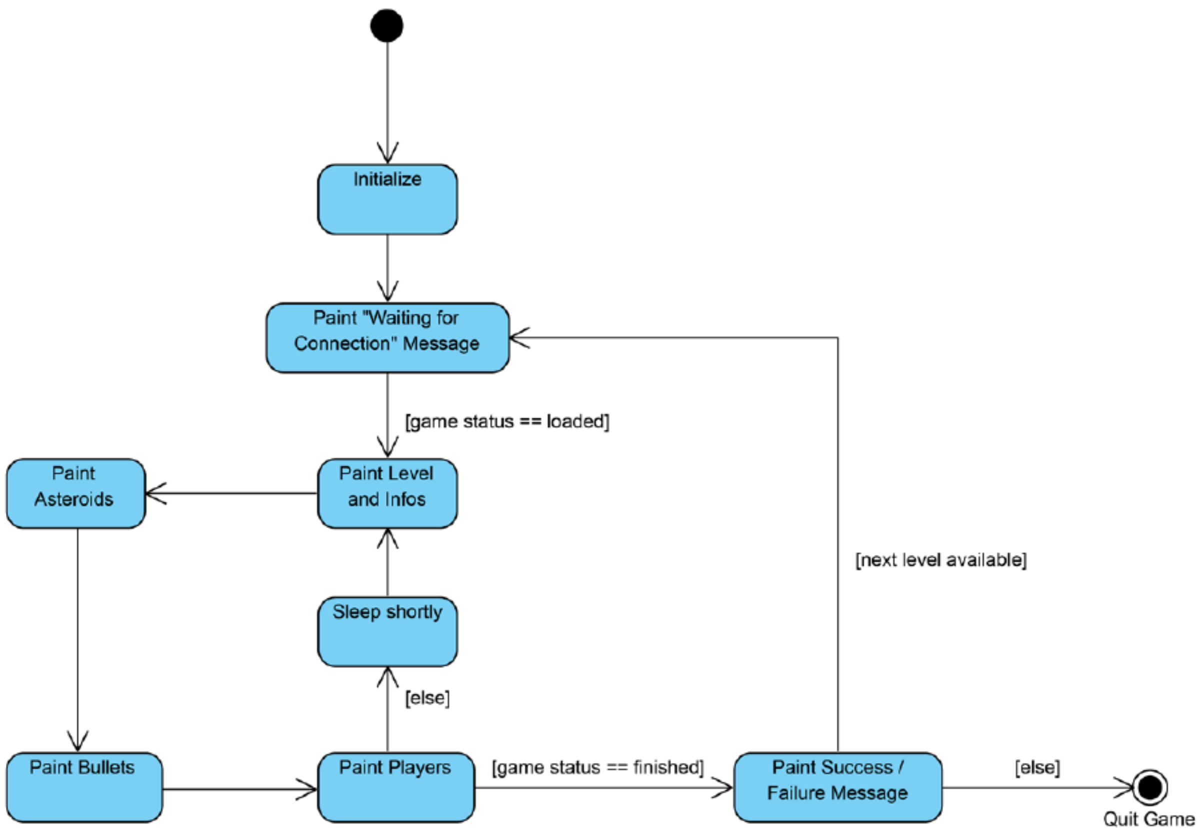
BEHAVIORAL MODEL OF THE GAME INITIALIZATION: UML


sd Client
Client1 : Game Logic
Client1 : Logic Connector
Client1 : Ships
Client1 : Status
Client1 : Data Store Connector
Client1 : Client To Server Connector
1: request initialization
6: forward notifications
7: request state updates
8: forward requests to data store components
9: forward update request
10: notify about status update
11: forward notification
12: forward notification
13: request ship creation
14: forward request
15: forward request
16: notify about new ship
17: forward notification
18: forward notification
23: forward notification to all other clients


sd Server
Server : Server to Clients Connector
Server : Data Store Connector
1: request initialization
2: request currently stored state
3: forward request to data store components
4: notify about current states
clock, status, level, ships, asteroids, bullets
5: forward notifications
6: forward notifications
18: forward notification
19: request data store update: new ship
20: forward request to data store components
21: notify about data store update: new ship
22: forward notification
23: forward notification to all other clients

The above sequence diagram (splitted for better display) shows the client initialization process. After a connection to a server has been established, the client requests the initialization data, which the Server to Clients Connector obtains from the data store and sends back to the client. The Client to Server Connector then requests data store status updates locally, which in turn trigger the creation of a ship for the player (initiated by the Game Logic). The notification about the new ship will be serialized and transferred to the server by the Client to Server Connector. This process has been modeled exemplarily for other kinds of state exchange between the clients and the server.

BEHAVIORAL MODEL OF THE GUI FUNCTIONALITY: UML



The above state machine diagram illustrates the GUI rendering process. After initialization of the component, a message is shown to notify the user that the game is waiting for the connection to the server. Once the game is loaded, the GUI rendering process enters its main loop (paint level and information display, asteroids, bullets and players). After each cycle through the loop, the game status is checked and if the game has finished, the GUI paints a message notifying the user about the game outcome. When a next level is available, the GUI reenters the loop through the waiting-for-connection message state. Otherwise, the game is complete and will exit.

This diagram does not show any of the input processing that the GUI component has to implement as well. This is because the input processing can be implemented completely separately from the here depicted rendering process (in another execution thread).

CONSITENCY BETWEEN MODELS

All models are based on the concept of components and connectors. The xADL model functions as a reference model for all other models, as it describes the overall coarse-grained application architecture and all other models refine certain parts of this architecture. We will use it therefore to show the other models' consistency with this model.

In ACME, components and connectors can be modeled explicitly, similarly to xADL. They communicate through ports (components) and roles (connectors), which is little different to xADL's interfaces. The ACME models above define refinements of the client/server connectors in the xADL model. To allow modeling the substructure of the client/server connectors, they each had to be modeled as a component in ACME. Nevertheless, they provide and require the same interfaces / ports (Client to Server Connector: to TCP, UDP, Data Store, Thresholding; Server to Clients Connector: to TCP, UDP, Data Store). Thus, the refinement these models describe fits well into the xADL model.

The UML sequence diagram does not feature an explicit modeling of components and connectors as language-level types. Thus, they have been modeled as common actors in the sequence diagram and named respectively. The sequence diagram only shows message / event passing in the C2 request/notification style, and only between components and connectors that have connected interfaces. The granularity of the interactions / component boundaries corresponds to the granularity in the xADL model and abstracts from the refinements in the ACME model (which does not pose any inconsistency, as the sequence diagram shows another level of abstraction and could be refined to provide this information as well). Thus, the models are consistent.

The UML state diagram shows the refinement of the operation of a single component in the xADL model as state transitions of the component (behavioral, not structural) and thus does not need to provide support for component or connector notations. It conforms to the xADL structure in that it only supposes knowledge about information that the GUI component could acquire through notifications from the data store.

MODELING ASSESSMENT

GENERAL ASSESSMENT OF SOFTWARE ARCHITECTURE MODELING

As main take-away from this exercise, I consider the insight that modeling architecture can be long, painful process, but also a fruitful process, even for a game of this small scope. The explicit modeling makes you think carefully about the features, necessary component boundaries and relationships, and thus can reveal thought problems that otherwise would have only surfaced someway into the implementation. In this case, for example, the modeling forced me to think of a way to model the movement of items in the game in a way that was easily transferrable over the network without causing too much traffic, thus preventing a possible less thoughtful implementation. It was definitely a worthwhile experience.

Nevertheless, I am unsure whether architecture (at least in the requested formal level and detail) is a good investment for a project like this (in "real" life), as for a project of this scope, apart from the learning effect, the only take-away from the architecture is the possible speedup in the implementation gained through a well-defined architecture. Other benefits of software architecture will not be used (for example, documentation for later project development/maintenance, component reuse, analysis or similar). Thus, the cost of going the wrong way in the implementation (for example, as usually pursued in extreme programming techniques), might not be outweighed by the speedup gained through the use of architecture models (given the big amount of time it takes to create the architecture). I stay skeptical, and look forward to the insights gained through the implementation of the game based on the architecture models.

LESSONS LEARNED DURING THE MODELING PROCESS

Looking back on the obstacles I had to deal with and insights I gained during the modeling process, the first thing to note is probably that this assignment had a very slow and difficult start. Especially if you aren't very familiar with real time networked game development on the one hand and the architecture tools on the other hand, being productive from the start was very hard. Without a specific idea about the implementation architecture, you cannot really start modeling. So apart from learning about the tools, you also need to start by researching common problems regarding design decisions with networked games. Thus, it was a very time-consuming task.

Apart from this, another obstacle was finding ways to make the different modeling notations compatible (i.e. enforcing the same component and connector boundaries in all languages). For example, the hierarchical composition of components in ACMEStudio did not easily allow connecting connector roles to ports of outer components (see tools section). Also, maintaining the consistency between the models in the different tools was difficult, as this was only possible by manual inspection and change. This definitely created desire for the idea of modeling all different aspects and diagram types in a single tool that can make syntactic and semantic crosschecks between them (e.g. references to same entities in all models) and possible even allow changes in one model to propagate to other models.

MODELING LANGUAGES

xADL 2

Comparing xADL with the other languages, the indifference between interaction of connectors and components in xADL (both have a set of interfaces and can be connected to either type) made a positive impression. xADL provided a direct notation of components and connectors, thus making it easy to put the architectural thoughts (which, in my case, conformed to hierarchical components and connectors) into a formal model. On another note, I found the types / structures concept of xADL 2 very intuitive for creating substructures, even though it was removed in xADL 3. Furthermore, one can safely say, that the actual XML representation of the model cannot be edited without tool support like Archstudio (the final document was about 4000 lines long). The extensibility features of xADL were not used for this project, they simply were not necessary for its scope.

ACME

ACME provides an easy and readable textual syntax, and models can be easily created without tool support. Nevertheless, there is a graphical editor for ACME included in ACMEStudio, which makes model creation even more comfortable. The language provides extensibility features, including adding properties to component / connector types and style families to create new types of model entities. The editor also provides the necessary configuration points for their representation (see next section).

For this project, the entity set did not need to be extended, but one feature that was missing in the default language entities was the direction of the ports (in / out / inout) and had to be modeled as property. Those sadly cannot easily be displayed in the visual representation.

UML

The UML sequence diagram does not notationally support the architecture concept of components and connectors, which immediately shows the fact that this type of diagram was not specifically designed for architecture modeling in the granularity I used it for. Nevertheless, the loosely defined syntax and semantics of UML sequence diagrams allowed me to find a way to model everything I expected. Components and connectors were both modeled in the same way, the message passing semantics provided enough room to support the coarse-grained level I used, and even modeling messages to several components at once could be modeled with lost and found messages. Nevertheless, it would have been nice to have syntactical support for all of the above, so as to reduce possible misinterpretation of the diagram representation.

The UML state diagram could be easily used to refine a single components behavior, as no interference with other components is explicitly modeled in it. While this may be an advantage on the one side, on the other it also is a disadvantage because inconsistencies between the models (for example, the use of data in the state diagram transitions that cannot be obtained through the static/structural architecture) cannot easily be checked for and even manual inspection requires effort.

Concluding, UML allowed a very fast modeling because of few syntactical constraints, and the wide usage of UML in the industry comes with very good tool support. Its usage can serve the purpose of building a consensus of the understanding of the architecture, but it requires explanation and discussion of the models to obtain it, given the ambiguity of its notations regarding architectural models.

MODELING TOOLS

Archstudio

I decided to use Archstudio 4 for the modeling in xADL. There were several factors that contributed to this decision. In the lecture, this version was promoted as more “stable”, and supporting the 1.x mapper tools (which I would like to try out). Also, I gave Archstudio 5 a fast shot, and couldn’t figure out how to do the hierarchical compositions without component types at once (I didn’t realize the drag and drop features and could not find any context menu entries for this in Archipelago neither any how-to that would explain this. Personally, I found working with the types concept very intuitive in Archipelago in Archstudio 4, although mapping signatures and interfaces was a little cumbersome because it felt like defining them twice). Thus I went with the older version, especially as it seemed to still be well maintained and working on top of Eclipse Juno. All experience described below refers to Archstudio 4.

The context-menu based interaction in Archipelago was easy to understand, although a full tutorial on hidden features would have been nice. I would have loved some keyboard shortcuts (for example, F2 for renaming entities, or some key combination for creating new interfaces on components) and more mouse actions (for example, ctrl-click-and-drag or something similar for creating connections between interfaces). I experienced some issues with drawing exactly vertical/horizontal links, which were promptly fixed on request. Other than this, I would like to propose an improvement for changing interface directions: If two interfaces are connected, changing the direction of one interface could possibly also change the direction of the other interface, as the connection would not make much sense otherwise. For example, if one interface is changed to “in”, the other end of the connection should be an “out” interface. Of course, if more than one connection to the interface exists, other possibilities might have to be considered. Another improvement, which is probably only relevant for the typing concept of xADL 2, is to add options for creating interfaces from the signatures of the component types or vice-versa. Something else I really missed was undo/redo functionality in Archipelago.

Despite all these possible improvements, Archstudio and Archipelago were quite easy to get comfortable with and very stable. My overall impression was a positive one.

ACMEStudio

As mentioned before, the editor in ACMEStudio provides a very good configurability of the representation of entities and entity types. For example, it was easy to add a label to the connector representation. Especially given the extensible nature of ACME, this seems like a nice feature. Another positive impression was the easy possibility to switch between source code and the graphical editor through editor tabs, although the source code representation inside the editor was sadly very buggy as soon as component hierarchies were introduced. Which introduces the first and foremost issue with ACMEStudio: It is buggy and has a very low responsiveness (it is slow!), especially if the model gets a little larger. Also, the graphical editor did not allow binding interface roles to outer component ports in a hierarchy (as described above), although the language supports this.

Concluding, I would like to say that even though the website for ACMEStudio seemed only half-way filled, there were nice PDF tutorials available which provided a good introduction and made using the editor feel relatively intuitive – I did not have to search for long for features. I would not use the editor for any productive projects, but it showcased a few nice ideas worth exploring.

Visual Paradigm

I should say that I had experience in working with Visual Paradigm before, and thus my impression might look better than it once was. I can remember that I had a very fast start with this tool when I first used it. The controls are very intuitive and fast. Most actions can be accessed via buttons that show up on mouse-over / selection of entities in the diagram directly next to the entity. Making connections between entities by drag-and-drop of those buttons is very simple. Some features are a little hidden in property dialogs; for example, how to change the way that message sends are numbered in a sequence diagram was not easy to find out. Also, the tool is not bug-free either – for example, activity markers on a sequence diagram lifeline happen to change length sporadically without apparent reason. Nevertheless, one can easily recognize that Visual Paradigm is a commercially deployed product, and has matured well.

GAME / SYSTEM PROPERTIES

As already mentioned before, one big thing I learned about the properties of the system was how to model the state of the moving items in the game in a way that was easily transferrable over the network. Apart from this insight, modeling the client-server connector's inner structure showed that both a UDP and a TCP connection between each client and the server would be advisable. Through the model of the game initialization phase, I could clarify the relationships between the individual components and the way information would get from the clients to the server and vice-versa.

REFERENCES

[1] Atari “Lunar Lander” online arcarde game. http://atari.com/arcade#!/arcade/lunarlander/

[2] Archstudio 4. http://www.isr.uci.edu/projects/archstudio-4/www/archstudio/

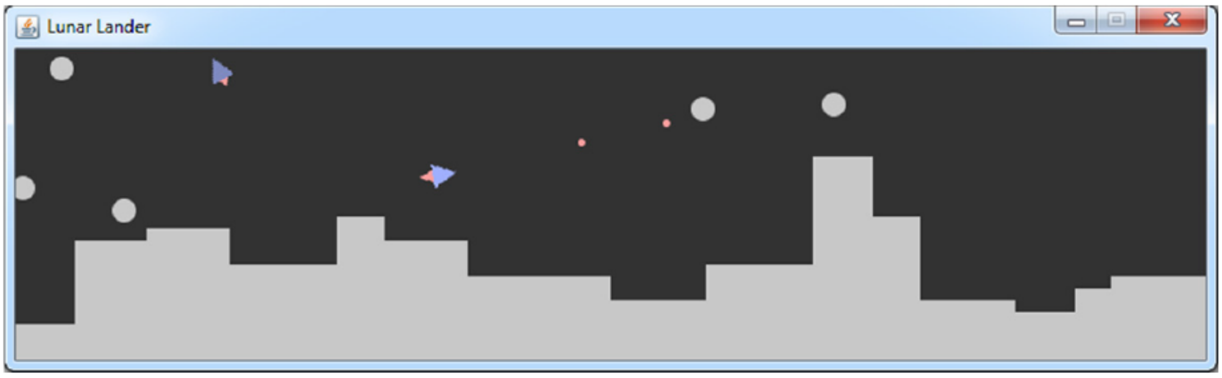
[3] ACMEStudio. http://www.cs.cmu.edu/~acme/AcmeStudio/index.html

[4] Visual Paradigm for UML. http://www.visual-paradigm.com/product/vpuml/

[5] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. 2009. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.

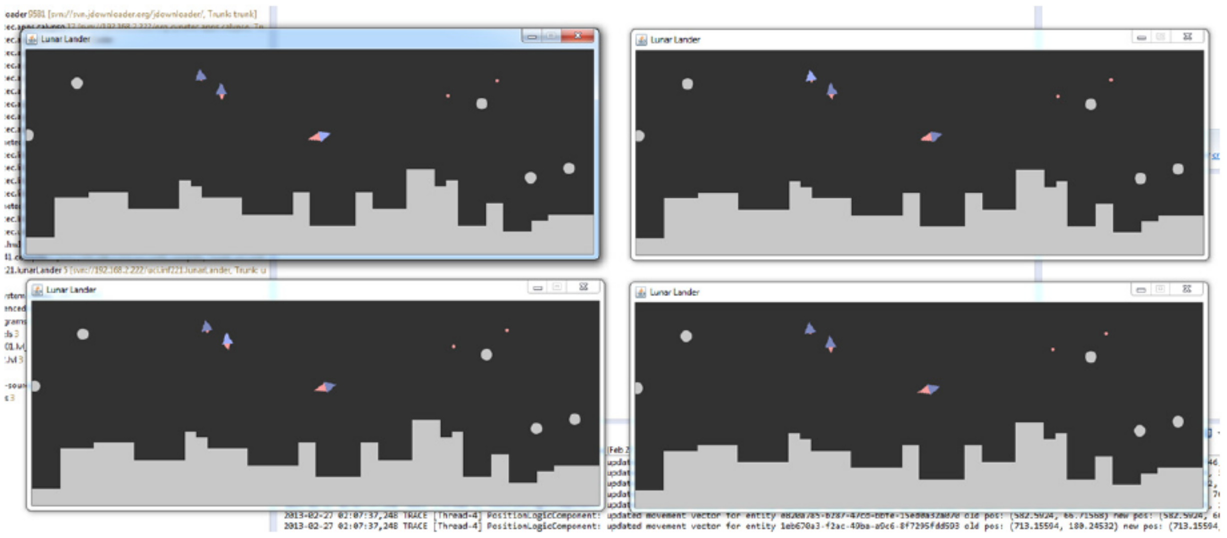
RUNNING GAME

Before going into the details of the implementation and architecture changes, I would like to give you a short impression of the current version of the game. A short review of the game concept (see Section “Revised Architecture” for a complete game concept description): Each player gets to steer his own ship, ships can fire bullets to destroy other ships or asteroids, which the ships have to escape from while pursuing the game goal of one player’s ship safely landing on the moon surface.



The above screenshot shows the game in action. To distinguish between local and remote player ships, the remote player ships are drawn slightly transparent. In the situation above, the active player has shot two bullets on the asteroids arriving from the right. The GUI is simplistic, but well-performant. Players cannot collide with each other, but can shoot each other. There is currently no fuel limit posed on the players. The game allows an easy addition of levels through text files containing a matrix representing the level (wherein a character of “.” denotes a sky block of size 8x8 pixels, while “x” denotes a moon surface block). Once a game is started, the available levels will be played in order until all have been completed successfully.

The following screenshot shows the game with 4 clients connected from the same physical machine at the same time. It only shows three ships, as the lower-right client’s ship was already destroyed.



```
2013-02-27 02:07:37,248 TRACE [Thread-4] PositionLogicComponent: updated movement vector for entity e820a785-b287-47c0-bbfe-15e0da32a870 old pos: (582.5924, 66.71568) new pos: (582.5924, 6
2013-02-27 02:07:37,248 TRACE [Thread-4] PositionLogicComponent: updated movement vector for entity 1eb670a3-f2ac-40ba-a0c6-8f7295fdd593 old pos: (713.15594, 180.24532) new pos: (713.15594,
```

CHANGES TO THE GAME ARCHITECTURE

Of course, the implementation phase did not go without (minor) changes to the architectural models. Thus, in this section, I describe the changes applied to the earlier proposed architecture. The complete revised architecture (including the unchanged game concept) is included for your convenience in the next section.

The biggest change in the structure of the game is that handling the destruction of asteroids and bullets was moved from the server to the clients, making the Bullet Manager in the server obsolete, while the Asteroid Manager in the server architecture now only is responsible for the (pseudo-random) creation of asteroids. This change was done during implementation, because the whole collision detection (including those of asteroids and bullets) was done by the clients anyway, and thus, the clients already initiated asteroid destruction requests on collisions. This made it seem only logical for the client to be responsible for destroying asteroids and bullets on out-of-world movement, too. A result of the clients initiating the destructions is that it is possible that several clients initiate the same collision at the same time and thus forward it to the server several times. This has of course been regarded during implementation and was not considered a problem. The duplicate calculation (each client does the same calculations) and additional communication does not outweigh the advantage gained by the clients being responsible for the collision detection: As each client is responsible for the collisions of his own ship (only the player's client can destroy its own ship), the network latency does not affect the steering of the ship and its collision detection. (Although this poses security concerns, which would have to be addressed for a public version of the game, as a modified client could allow its ship to be indestructible.)

In order to have a consistent time base across server and clients, a time synchronization mechanism between server and clients was introduced. Therefore, the server acts as an NTP server (functionality provided by the Client Connection Manager Component), and the Clock Component as an NTP client. The structural models have been updated correspondingly.

Another structural change is the addition of a Level Manager component to the server architecture. This component is responsible for changing the level status to a ready-to-start state once enough clients have connected (the minimum player number is adjustable) and for the change of levels after a level was completed (or restarting it on failure). Also, the game/level status (initializing, loaded, ready, started, failed, success, error) is stored in the Level Component, instead of the Status Component, as it is a level-specific status. The Status Component on the other hand could store level-spanning game statistics, e.g. player scores (although such a feature has not yet been added in the implementation, and thus, the Status Component does not store anything yet).

As the level status is stored in the Level Component, the behavior of the client initialization was changed to reflect that the game logic creates a ship based on the level status change notification sent by the Level Component. Another minor change to the client initialization involves that the client does not explicitly request the initialization data but instead does so implicitly by connecting to the server.

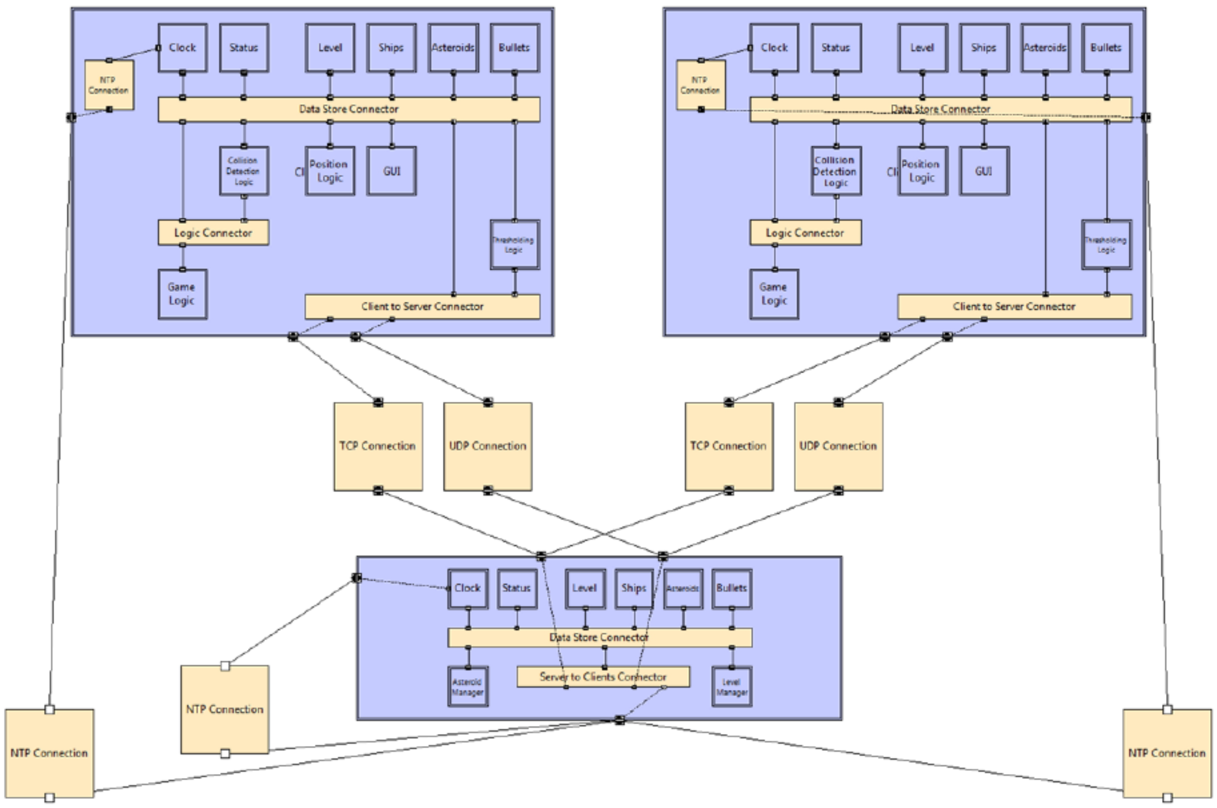
Last but not least, the role of the Client States Component in the Server to Clients Connector of the server architecture was changed in that it also stores the client/player-to-ship associations. This information is used by the Connection State Change Handler to destroy the client's ship if a client disconnects during a game. The client-to-ship associations are updated by the Client Message Handler Component whenever a client notifies the server about a new ship creation.

REVISED GAME ARCHITECTURE

GAME CONCEPT

The presented game is a simple multi-player variant of the 2D game Lunar Lander[1]. Each player controls a separate ship. As additional obstacle, the players have to escape asteroids that fly over the moon surface. Their ships are also equipped with firearms, which can be used to destroy asteroids. Instead of fighting against each other, the players cooperate towards the goal of one of them safely landing on the moon surface. This could e.g. be accomplished by other players watching out for asteroids and clearing the way while one player tries to land. Players can shoot each other, but if one player's ship is destroyed, the remaining ones continue the game without that one, either until one ship successfully landed, or until all ships were destroyed.

COARSE-GRAINED STRUCTURAL GAME MODEL: XADL



The game is divided into a client and a server application. The above diagram shows two of clients connected to a server, all of them with their internal component structure. The game architecture does not pose any explicit limits on the number of clients simultaneously connected to the game server.

The connection between a client and a server is maintained over two channels / connections: a TCP connection and a UDP connection. While the UDP channel is used on the one hand to announce server availability (via broadcasts by the server) and on the other hand to exchange less important information (for example, ship movement updates which do not require a retransmission in the event of packet loss, because they will become obsolete with the next update), the TCP connection is established to monitor the client-server connection status and exchange important information (for example, ship or asteroid destructions and status changes).

Both the client and the server implementation are based on the C2 architectural style[2] on the displayed granularity level, although the implementation of the single subcomponents (for example, GUI or Game Logic) does not necessarily conform to this style as well.

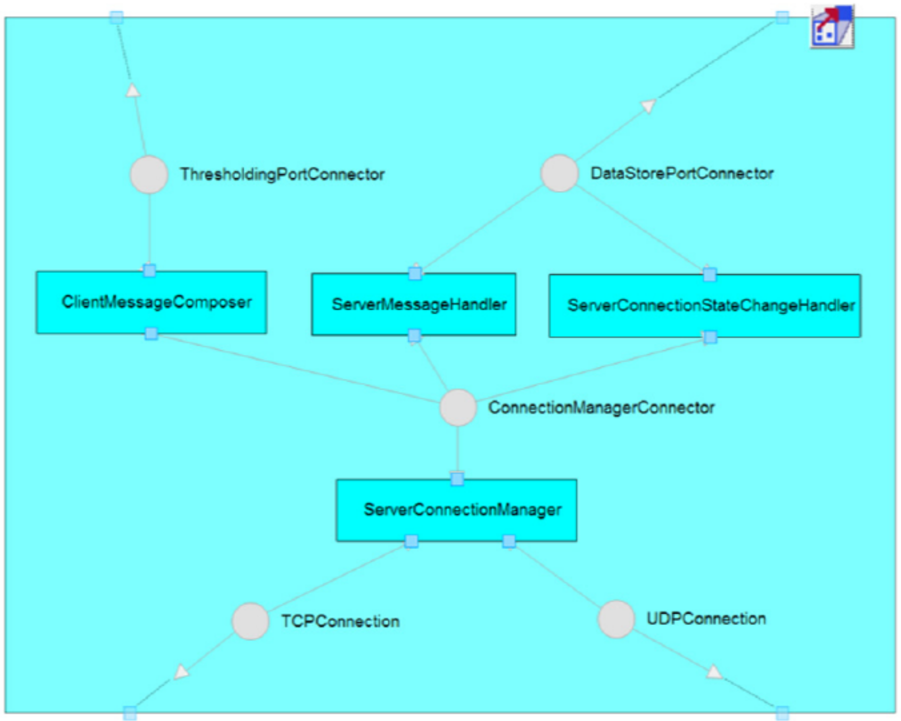
The uppermost level in both the client and server architecture consists of data store components (Status, Level, Ships, Asteroids, Bullets) and the game clock component. All of which are connected to the same connector (Data Store Connector). The data store components can process data store update requests from the components below them in the architecture, and will send data store update notifications downwards once they performed an update. The movement and position of moveable items of the game (ships, asteroids and bullets) is stored by a base position and speed vector for a certain time and an acceleration vector, to decrease the necessary network traffic for position / movement changes.

In the server architecture, the Data Store Connector connects the data store directly to the connector that handles the client-server communication (Server to Clients Connector), as well as to the components that manage the Asteroids (creation) and the Level (loading/changing). Other than that, the server mainly functions as a data store only, forwarding requests and notifications between the clients and managing the client connections and their data consistency. (For some more details see the game initialization model further below).

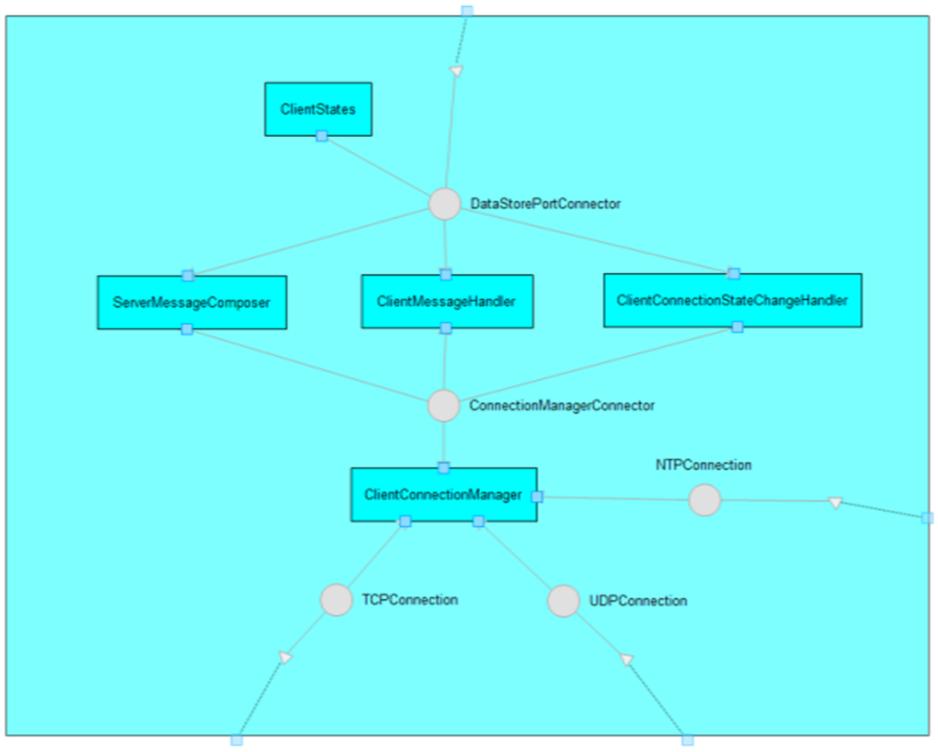
In the client architecture, the Data Store Connector connects to the Collision Logic, Position Logic and GUI components to the bottom, as well as to the Logic Connector and the Client to Server Connector, whereby update notifications to the Client to Server connector are first processed by a Thresholding component before they are passed on to the connector. The Position Logic calculates the current position and speed of ships, asteroids and bullets, by extrapolating from the last position of the items and their last speed and current acceleration vectors and writes the results back into the data store (by sending the respective update requests). The GUI component paints the user interface based on the data store state and sends update requests to it based on the user input (for example, changes in the ship acceleration or new bullets). The Thresholding component is responsible for only passing on data store changes to the Client to Server Connector when the change is bigger than a given threshold for the specific type of the change (for example, only pass on ship movement changes if the movement deviates significantly from the predicted path from the last sent position, speed and acceleration vectors). Collisions between Bullets, Asteroids or Ships will be detected by the Collision Detection component and passed on to the Logic Connector, which in turn passes on notifications from the data store and Collision Detection to the Game Logic component. This component will react upon possible collisions or state changes and evaluate their effect, issuing possible data store update requests.

In order to have a consistent time base across server and clients, the server acts as an NTP server (functionality provided by the Client Connection Manager Component), and the Clock Component as an NTP client. The Clock component of the data store in the server also connects to the NTP server of the server, so as to allow a common implementation for both client and server.

STRUCTURAL MODELS OF THE NETWORK CONNECTORS: ACME

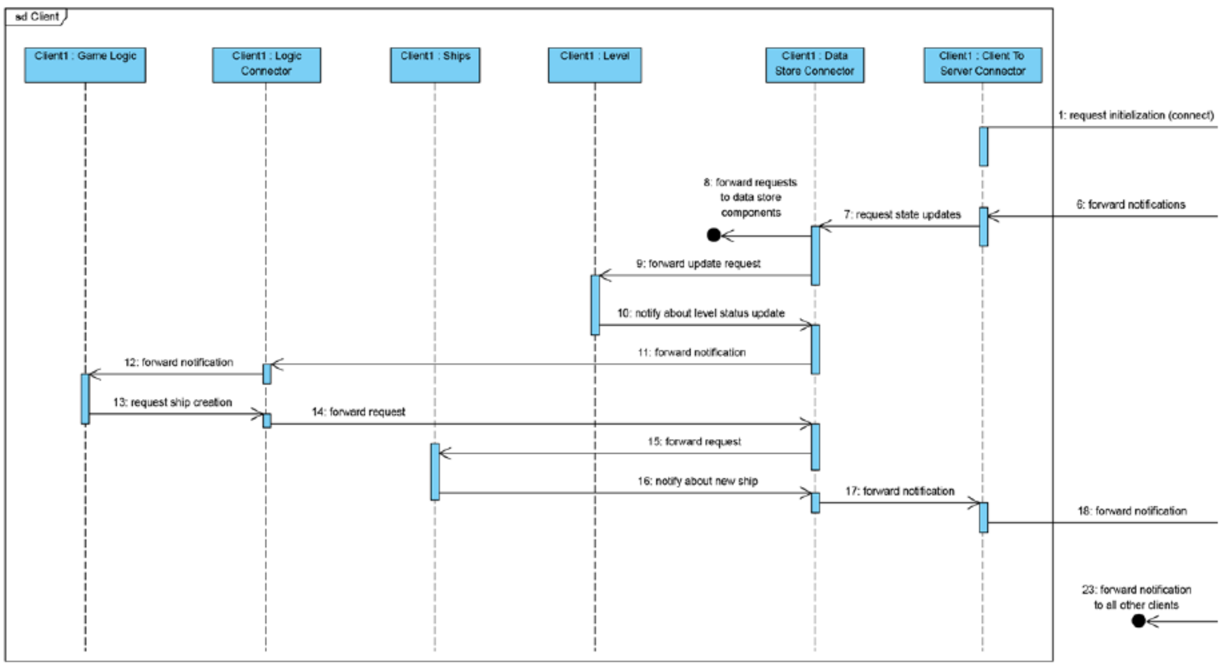


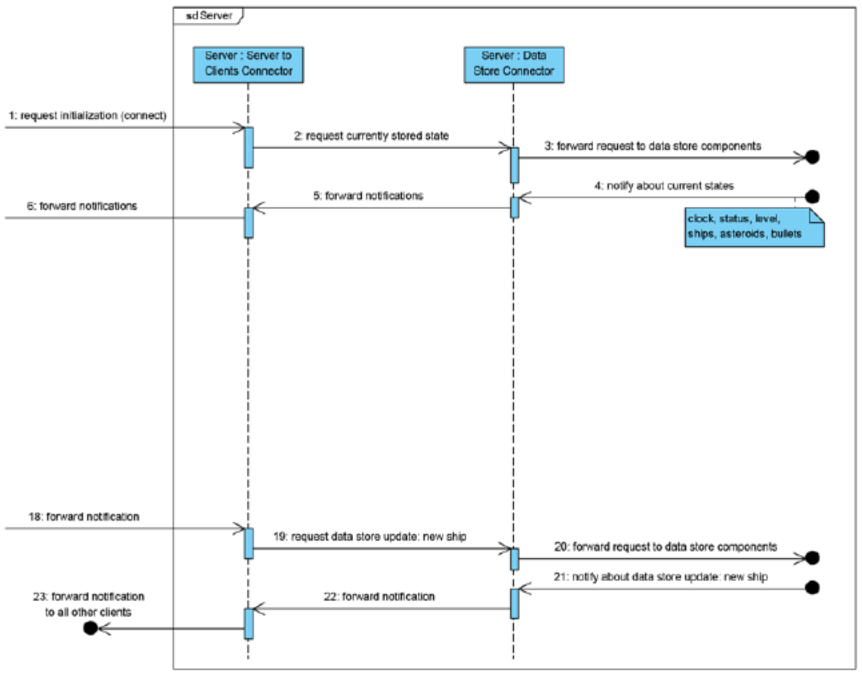
The above diagram shows the internal structure of the Client to Server Connector present in the client architecture as shown in the xADL model. Its architecture also follows the C2 style. The Server Connection Manager is responsible for serializing and deserializing messages, and transmitting / receiving them over the TCP and UDP connections to / from the server. In the case of arriving messages, it sends a handle message request to the Connection Manager Connector, which will forward it to the other three components. The Server Message Handler will then analyze the message and issue data store update requests to the Data Store Port Connector, which forwards them to the Data Store Connector. If the connection status changes, the Server Connection Manager requests the handling of the state change, which will be served by the Server Connection State Change Handler. For example, if the Server Connection Manager signals a connection loss, the state change handler will request a game status update respectively. Contrary, if any data store update notifications arrive at the ThresholdingPortConnector, then those will be handled by the Client Message Composer and the Server Connection Manager Connector will be notified of a new message ready for transmission (which the Server Connection Manager then sends).



The above diagram shows the detailed architecture of the Server to Clients Connector present in the Server architecture in the xADL model. It is analog to the Client to Server Connector architecture depicted above, but includes an additional Client States component, which stores the current connection states and player/ship associations of the clients. These are updated by the Client Connection State Change Handler (connection states) and the ClientMessageHandler (player/ship associations). The Client Connection State Change Handler can use the notifications about their change (sent by the Client States component) to handle the loss of client connections by destroying their associated ships. Theoretically, the Server Message Composer could use the notifications to broadcast the client states to all clients, but in the current implementation of the game, there is no need for the clients to know about other client states. The Client Connection Manager has similar responsibilities as the Server Connection Manager in the Client to Server Connector architecture, but handles more than one connection simultaneously and provides a TCP connection / socket that the clients can connect to. The Client Message Handler evaluates the incoming messages and requests data store updates. The update notifications will be handled by the Server Message Composer and broadcasted to all other clients.

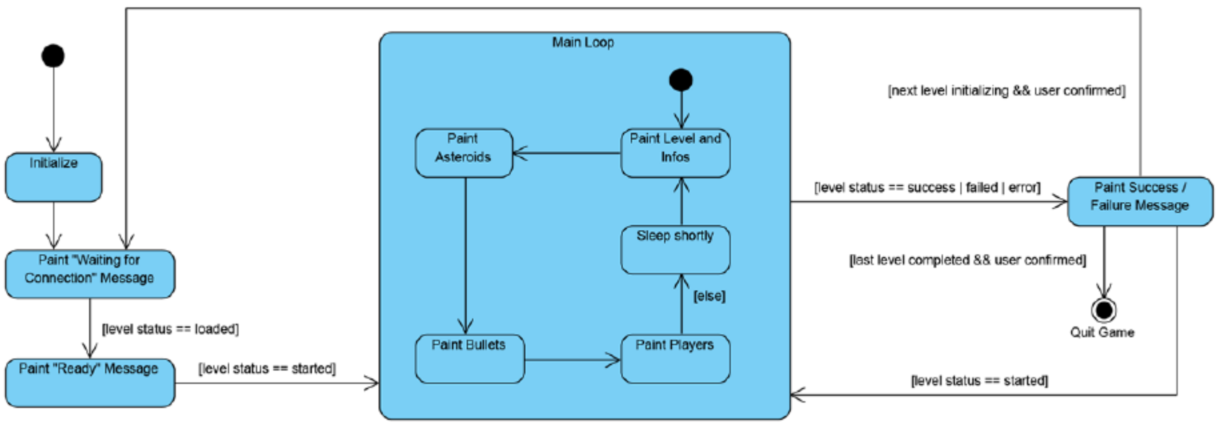
BEHAVIORAL MODEL OF THE GAME INITIALIZATION: UML


sd Client
Client1 : Game Logic
Client1 : Logic Connector
Client1 : Ships
Client1 : Level
Client1 : Data Store Connector
Client1 : Client To Server Connector
1: request initialization (connect)
6: forward notifications
7: request state updates
8: forward requests to data store components
9: forward update request
10: notify about level status update
11: forward notification
12: forward notification
13: request ship creation
14: forward request
15: forward request
16: notify about new ship
17: forward notification
18: forward notification
23: forward notification to all other clients


sd Server
Server : Server to Clients Connector
Server : Data Store Connector
1: request initialization (connect)
2: request currently stored state
3: forward request to data store components
4: notify about current states
clock, status, level, ships, asteroids, bullets
5: forward notifications
6: forward notifications
18: forward notification
19: request data store update: new ship
20: forward request to data store components
21: notify about data store update: new ship
22: forward notification
23: forward notification to all other clients

The above sequence diagram (splitted for better display) shows the client initialization process. By establishing a connection to the server, the client implicitly requests the initialization data, which the Server to Clients Connector obtains from the data store and sends back to the client. The Client to Server Connector then requests data store status updates locally, which in turn trigger the creation of a ship for the player (initiated by the Game Logic). The notification about the new ship will be serialized and transferred to the server by the Client to Server Connector. This process has been modeled exemplarily for other kinds of state exchange between the clients and the server.

BEHAVIORAL MODEL OF THE GUI FUNCTIONALITY: UML

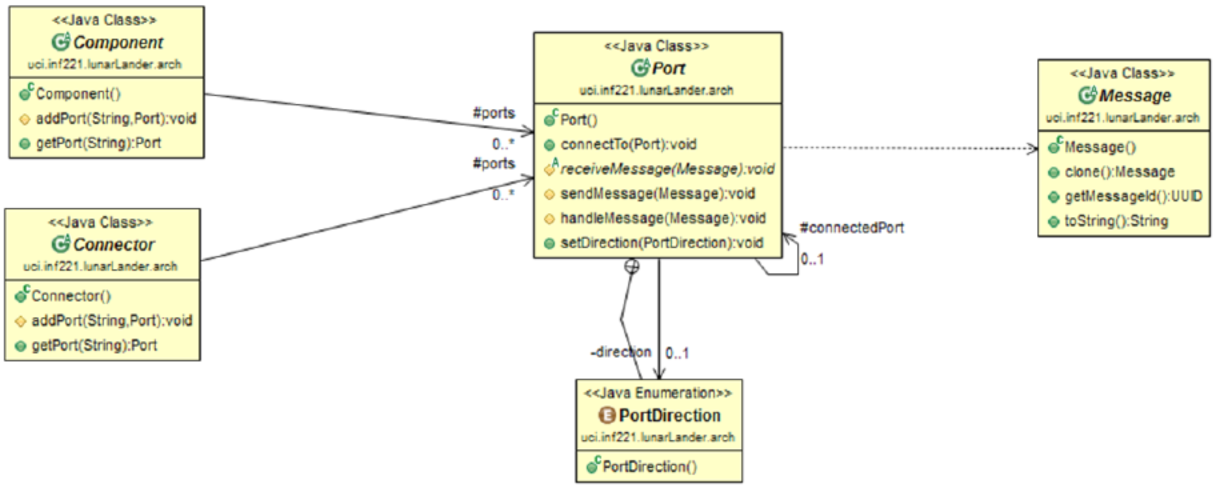

Main Loop
Initialize
Paint "Waiting for Connection" Message
[level status == loaded]
Paint "Ready" Message
[level status == started]
Paint Asteroids
Paint Level and Infos
Sleep shortly
[else]
Paint Bullets
Paint Players
[next level initializing && user confirmed]
[level status == success | failed | error]
Paint Success / Failure Message
[last level completed && user confirmed]
Quit Game
[level status == started]

The above state machine diagram illustrates the GUI rendering process. After initialization of the component, a message is shown to notify the user that the game is waiting for the connection to the server. Once the game is loaded, a ready message is displayed (which can be confirmed by the user to start the game). After one of the clients started the game, the GUI rendering process enters its main loop state (paint level and information display, asteroids, bullets and players). If the game status changes to a finished state (completed, failed or error during game), the GUI exits the main loop state and paints a message notifying the user about the game outcome. After confirmation of the result message, the GUI either reenters the loop through the waiting-for-connection message and ready message states if a next level is available, or will exit (as otherwise the game is complete). When another client starts the next level before the local user confirms the result message, the game reenters the main loop directly from the result message state.

The diagram does not show any of the input processing that the GUI component has to implement as well. This is because the input processing is implemented completely separately from the here depicted rendering process (in another execution thread).

IMPLEMENTATION

In order to implement the described game, I built a simple Java class model to capsulate the architecture in, which is depicted below. It allows the explicit modeling of Components, Connectors and their communication through ports and messages. Each component or connector can have several ports, each port has a direction (in/out/inout) and is connected to at most one other port, which it can receive Messages from or send Messages to. All of these classes are abstract classes.



As this class model does not yet pose any restrictions on the connections between components and connectors, but the game is supposed to be implemented in a C2 style, the above model is extended with C2-specific classes for Components, Connectors, Ports and Messages. The generalization/specialization relationships between the C2-specific classes and the general model classes (from above) are shown in the class diagram below. Namely, C2Connectors and C2Components have BottomPorts and TopPorts, and can receive C2Notifications through C2TopPorts, C2Requests through C2BottomPorts and can send C2Notifications through C2BottomPorts and send C2Requests through C2BottomPorts. C2TopPorts can only be connected to C2BottomPorts and vice-versa. The C2Components and C2Connectors both implement specializations of the abstract C2Top/C2BottomPort classes, which forward their received messages to abstract method calls in the C2Component/Connector class, such that messages from any top or any bottom port are handled in the same way.

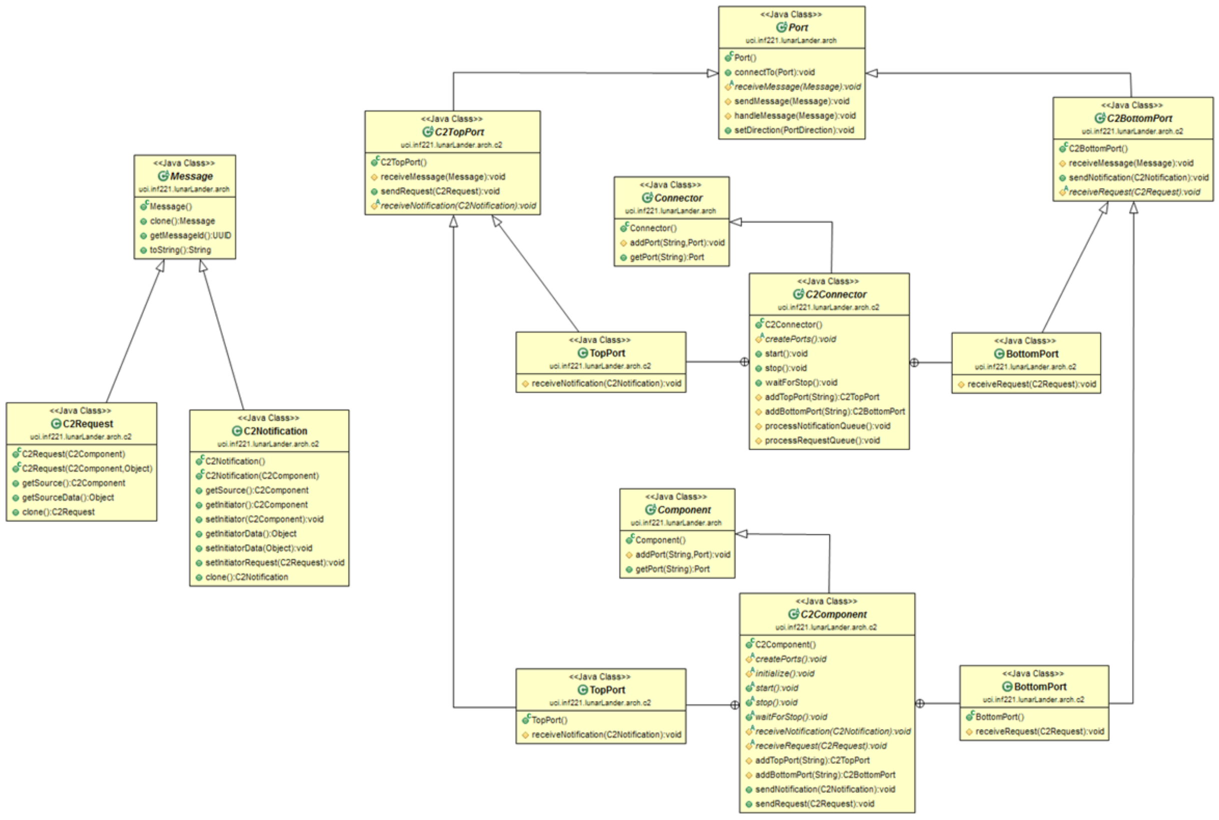
A few (more or less interesting) remarks on the implementation details:

The composition of the components and connectors happens in two places: each component or connector is responsible for creating its own ports. The connections between the ports of the components/connectors are then created from the “outside” in the startup routine of the client or server. Hierarchical compositions of components or connectors can be achieved by creating and connecting the components/connectors of the substructure in the initialization of the parent component/connectors and by exposing certain ports of the substructure components/connectors as ports of the parent component/connector.

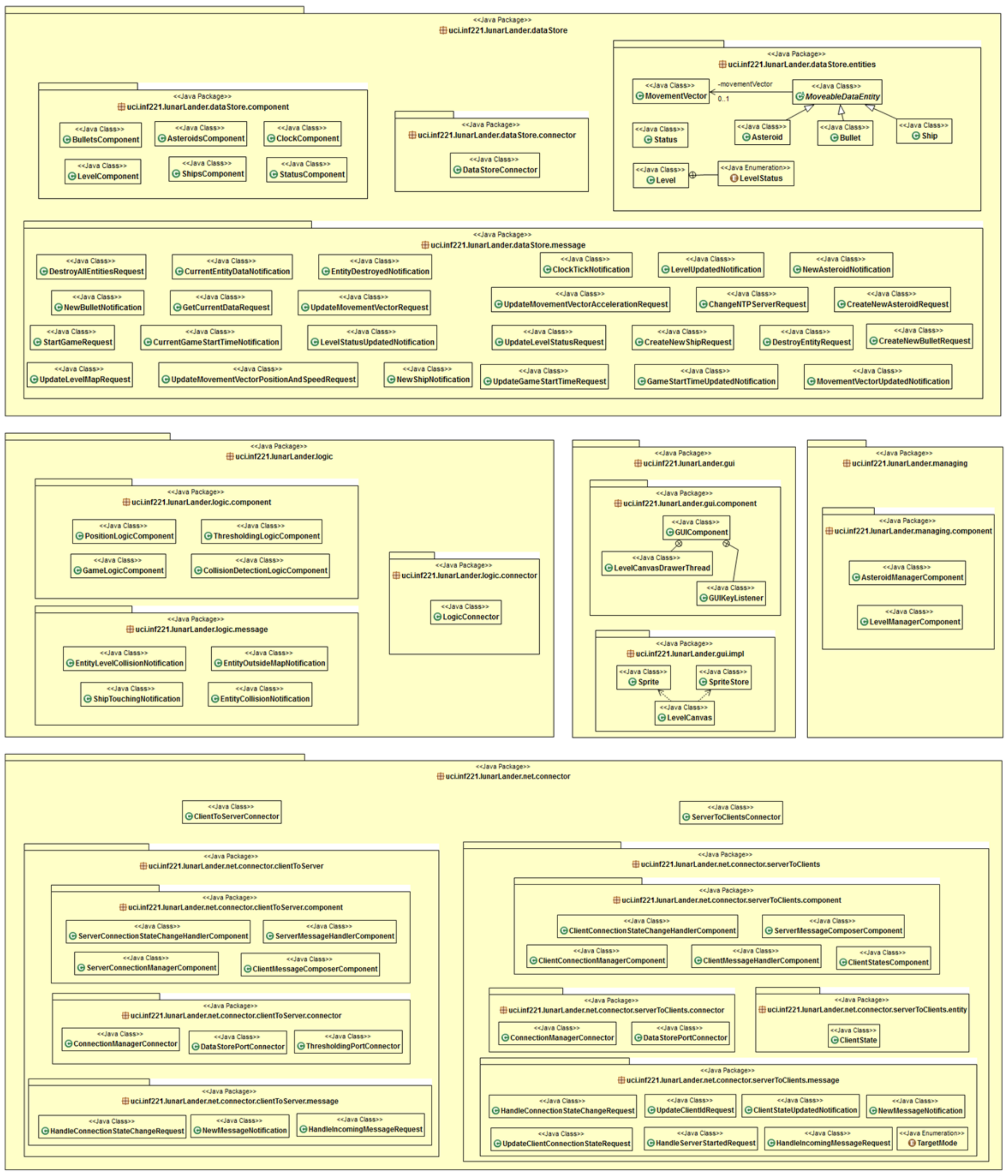
The ports directly forward the messages to the connected ports (via function calls). The C2Connectors are responsible for introducing an asynchronous decoupling of the components, which is achieved by queuing incoming requests and notifications in the connector for subsequent forwarding to the respective other “side” of the connector in other queue processing threads.

Both requests and notifications store their source component (the original sender of the message), and the notifications carry information about the sender of the request that caused their own issuance. This way, components can e.g. disregard notifications about game state updates they themselves caused.

The messages (in particular, the notifications) can be transmitted between client and server, and thus can be serialized. This also poses a new requirement that should be thought of: When a client forwards a message to the server to be broadcast to all other clients, it does expect the server to forward it to all other clients excluding it. Thus, the Server Message Composer needs to know which client the initial notification came from that caused the game state update, which it is being notified about. Therefore, the requests can carry a sourceData object and the notifications carry the sourceData object of their initiating request. The type and purpose of this object is not specified, as those should not be transparent to all components (explicitly, the data store components should not need to know of the network causalities). In practice, this means that the Client Message Handler component in the Server to Clients Connector sets the sourceData of the data store update request it issues to the client identifier that it received the message from. The data store component that handles the update request will set the initiatorData of the update notification it issues to the sourceData of the request. When the Server Message Composer receives a notification, it checks whether the initiatorData contains a client identifier, and if so, composes a message to all other clients excluding the client identified. A similar problem exists on the client side: If a client data store update was caused by a forwarded notification from the server, then the client should not forward the update notification from the data store about this update back to the server. Thus, the Server Message Handler component sets the sourceData of its data store update requests (and thus the initiatorData of the subsequent update notifications) to a value that the Thresholding Logic component can interpret to determine if the update was initiated remotely.



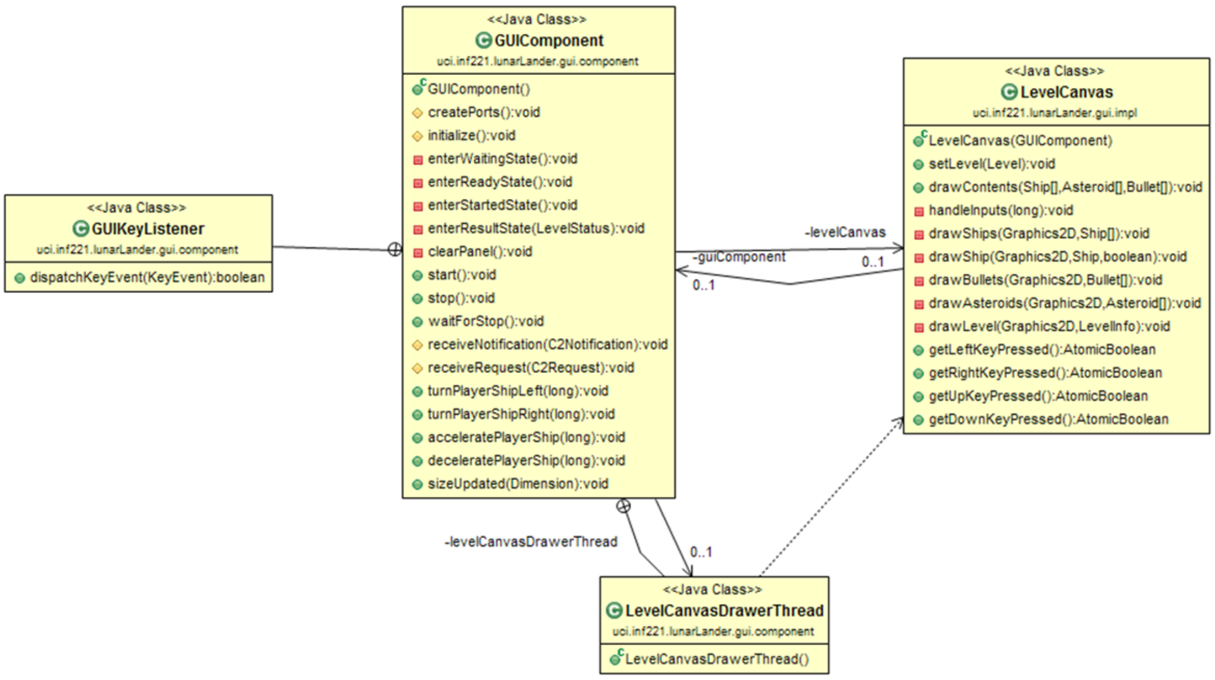
Based on this class model, I could implement the structural composition of the game components and connectors exactly and explicitly as described in the architecture models. Both client and server use the same Data Store Connector and data store components implementation, but connect different components on the bottom side of the connector. The package/class diagrams below illustrate the entities involved in the implementation.



uci.inf221.lunarLander.dataStore
uci.inf221.lunarLander.dataStore.component
BulletsComponent
AsteroidsComponent
ClockComponent
LevelComponent
ShipsComponent
StatusComponent
uci.inf221.lunarLander.dataStore.connector
DataStoreConnector
uci.inf221.lunarLander.dataStore.entities
MovementVector
-movementVector
0..1
MoveableDataEntity
Status
Asteroid
Bullet
Ship
Level
LevelStatus
uci.inf221.lunarLander.dataStore.message
DestroyAllEntitiesRequest
CurrentEntityDataNotification
EntityDestroyedNotification
ClockTickNotification
LevelUpdatedNotification
NewAsteroidNotification
NewBulletNotification
GetCurrentDataRequest
UpdateMovementVectorRequest
UpdateMovementVectorAccelerationRequest
ChangeNTPServerRequest
CreateNewAsteroidRequest
StartGameRequest
CurrentGameStartTimeNotification
LevelStatusUpdatedNotification
UpdateLevelStatusRequest
CreateNewShipRequest
DestroyEntityRequest
CreateNewBulletRequest
UpdateLevelMapRequest
UpdateMovementVectorPositionAndSpeedRequest
NewShipNotification
UpdateGameStartTimeRequest
GameStartTimeUpdatedNotification
MovementVectorUpdatedNotification
uci.inf221.lunarLander.logic
uci.inf221.lunarLander.logic.component
PositionLogicComponent
ThresholdingLogicComponent
GameLogicComponent
CollisionDetectionLogicComponent
uci.inf221.lunarLander.logic.connector
LogicConnector
uci.inf221.lunarLander.logic.message
EntityLevelCollisionNotification
EntityOutsideMapNotification
ShipTouchingNotification
EntityCollisionNotification
uci.inf221.lunarLander.gui
uci.inf221.lunarLander.gui.component
GUIComponent
LevelCanvasDrawerThread
GUIKeyListener
uci.inf221.lunarLander.gui.impl
Sprite
SpriteStore
LevelCanvas
uci.inf221.lunarLander.managing
uci.inf221.lunarLander.managing.component
AsteroidManagerComponent
LevelManagerComponent
uci.inf221.lunarLander.net.connector
ClientToServerConnector
ServerToClientsConnector
uci.inf221.lunarLander.net.connector.clientToServer
uci.inf221.lunarLander.net.connector.clientToServer.component
ServerConnectionStateChangeHandlerComponent
ServerMessageHandlerComponent
ServerConnectionManagerComponent
ClientMessageComposerComponent
uci.inf221.lunarLander.net.connector.clientToServer.connector
ConnectionManagerConnector
DataStorePortConnector
ThresholdingPortConnector
uci.inf221.lunarLander.net.connector.clientToServer.message
HandleConnectionStateChangeRequest
NewMessageNotification
HandleIncomingMessageRequest
uci.inf221.lunarLander.net.connector.serverToClients
uci.inf221.lunarLander.net.connector.serverToClients.component
ClientConnectionStateChangeHandlerComponent
ServerMessageComposerComponent
ClientConnectionManagerComponent
ClientMessageHandlerComponent
ClientStatesComponent
uci.inf221.lunarLander.net.connector.serverToClients.connector
ConnectionManagerConnector
DataStorePortConnector
uci.inf221.lunarLander.net.connector.serverToClients.entity
ClientState
uci.inf221.lunarLander.net.connector.serverToClients.message
HandleConnectionStateChangeRequest
UpdateClientIdRequest
ClientStateUpdatedNotification
NewMessageNotification
UpdateClientConnectionStateRequest
HandleServerStartedRequest
HandleIncomingMessageRequest
TargetMode

The Server to Clients and Client to Server Components are implemented as hierarchical C2 components as described in the architecture model. They make use of the Kryonet network library[3], which provides UDP and TCP connections between server and clients, as well as serialization/deserialization of java objects (i.e. our messages and their content) and a server discovery protocol that uses the UDP connection. This library is only used in the Server/Client Connection Manager Components in the hierarchical composition of the connectors, and not exposed to any other parts of the system.

The GUI Component uses the AWT toolkit [4] (included in the JRE) to draw a simple 2D representation of the level and the entities in it. The top-level states of the GUI component are implemented through state-enter methods, will are invoked depending on the current GUI state and incoming notifications. The main loop state starts a separate thread that calls a drawContents function on the canvas on which the level is painted. This function in turn calls methods which draw the level, asteroids, bullets and ships (in this order). Additionally, the main loop also processes arrow-key presses and calls the GUI component methods to turn or accelerate the player ship. The GUI structure is depicted in the following class diagram.


<<Java Class>>
GUIComponent
uci.inf221.lunarLander.gui.component
GUIComponent()
createPorts():void
initialize():void
enterWaitingState():void
enterReadyState():void
enterStartedState():void
enterResultState(LevelStatus):void
clearPanel():void
start():void
stop():void
waitForStop():void
receiveNotification(C2Notification):void
receiveRequest(C2Request):void
turnPlayerShipLeft(long):void
turnPlayerShipRight(long):void
acceleratePlayerShip(long):void
deceleratePlayerShip(long):void
sizeUpdated(Dimension):void
<<Java Class>>
GUIKeyListener
uci.inf221.lunarLander.gui.component
dispatchKeyEvent(KeyEvent):boolean
<<Java Class>>
LevelCanvas
uci.inf221.lunarLander.gui.impl
LevelCanvas(GUIComponent)
setLevel(Level):void
drawContents(Ship[],Asteroid[],Bullet[]):void
handleInputs(long):void
drawShips(Graphics2D,Ship[]):void
drawShip(Graphics2D,Ship,boolean):void
drawBullets(Graphics2D,Bullet[]):void
drawAsteroids(Graphics2D,Asteroid[]):void
drawLevel(Graphics2D,LevelInfo):void
getLeftKeyPressed():AtomicBoolean
getRightKeyPressed():AtomicBoolean
getUpKeyPressed():AtomicBoolean
getDownKeyPressed():AtomicBoolean
-levelCanvas
0..1
-guiComponent
0..1
-levelCanvasDrawerThread
0..1
<<Java Class>>
LevelCanvasDrawerThread
uci.inf221.lunarLander.gui.component
LevelCanvasDrawerThread()

To implement the NTP functionality, the server (specificly, the Client Connection Manager) uses a NTP Server implementation in Java (by me), which is based on the NTP packet types provided by the apache.commons.net packet [5]. The Clock Component creates an NTP client (provided by the same packet), once a connection to the server has been established (and before the game initialization process illustrated in the architectural model above begins).

Another interesting design decision made during implementation was to store the speed and acceleration vectors (of the MovementVectors) in polar coordinates. This preserves the direction of the vectors, even if their magnitude is 0, thus allowing the steering of the ship even if it is not accelerated without storing additional direction information.

The implementation contains a total of 115 classes and a little more than 6700 lines of code.

CONSISTENCY OF IMPLEMENTATION AND ARCHITECTURAL MODELS

From the above explanation of the implementation, it should be easily comprehensible that the structure of the implementation is consistent with the structural architectural models (xADL, ACME). The reason for this is that the explicit remodeling of the components and connectors, including the ports, directly represents the architectural models. The only ports that have not been implemented explicitly are the network connections (as those actually go beyond the scope of the game and require use of libraries). But as the network connections exist, so do these ports and connections implicitly. In order to convince you even more of the consistency between the structural models and the implementation, I have attached an extract of the source code that builds up the inner structure of the ClientToServerConnectorComponent (in the appendix) as an example for the described explicit modeling.

The behavioral model of the GUI as modeled in the UML state diagram was implemented as described above: The top-level states are implemented as states of the GUIComponent, while the main loop was implemented as successive calls to functions that draw the single entities. Thus, it is clear that the top-level states were implemented as modeled. For the main drawing loop, one can argue that the call of a function represents a state change as the execution control is given to another function. Thus, the internal states of the main-loop are implemented consistently as well. As the main-loop thread can be terminated at any point during the execution of the main-loop, the state change between the main-loop and the result state is implemented as modeled, as well.

Slightly more difficult task is to show that the sequence diagram of the client initialization is consistent with the implementation. A simple way to prove this for sure is to show you an excerpt of the log files of the client and server. Therefore, the C2Connector and C2Component classes have been instrumented to print a log file entry for every arriving and outgoing message. I removed part of the less important internal messages of the Client to Server / Server to Clients connectors, but the logs are still very long and can be found in the appendix.

What can be seen on the client side is that after the client connected to the server, it receives several CurrentEntityDataNotifications and a CurrentGameStartTimeNotification, which are translated into data store update requests by the ServerMessageHandlerComponent. Those update requests are forwarded through the DataStorePortConnector and DataStoreConnector to the single data store components, where their handling results in update notifications from the data store components to the DataStoreConnector. One of these is a LevelStatusUpdatedNotification, which is eventually received by the GameLogicComponent. This component in turn issues a CreateNewShipRequest, which is forwarded back to the data store, results in a NewShipNotification, which then is forwarded down to the ClientMessageComposerComponent. This component prepares it for sending to the server and passes it on to the ServerConnectionManagerComponent, which sends it to the server.

On the server side, one can see how the new client connection causes a GetCurrentDataRequest request to be sent to the data store, which the data store components reply to with CurrentEntityDataNotifications (most of which notify about an empty set of entities) and a CurrentGameStartTimeNotification (by the Clock Component). Those notifications are forwarded back down to the ServerMessageComposerComponent, which passes them on to the ClientConnectionManagerComponent to send back to the client. At some point, the ClientConnectionManagerComponent receives the NewShipNotification that the client sent out, which causes the ClientMessageHandlerComponent to issue a CreateNewShipRequest to the data store. The resulting NewShipNotification from the data store is finally forwarded to all other clients.

As this sequence of message exchanges corresponds to the (asynchronous) message transfers in the UML sequence diagram, this should suffice to show the consistency between the implementation and the behavioral model. With this, the consistency between all models and the implementation has been shown.

ASSESSMENT OF IMPLEMENTATION PHASE

GENERAL ASSESSMENT OF THE IMPLEMENTATION

The implementation of the game consisted basically of two more-or-less separate steps. First, the architectural structural model was transferred into the previously described structure of component and connector classes, and their composition. I chose to do this manually because of several reasons: I could see that the client/server communication based on the message-entities of the C2 architecture would require several extensions of the basic architectural elements. Also, I wanted to gain the experience and see how much work it actually was to explicitly model such a component/connector architecture as a class/object composition in Java. An alternative would have been to use an architecture/source code mapping tool (e.g. the 1.x-way mapper tools by Yongjie Zheng[6]).

The second step of the implementation was to implement the behavior of the single system components. This task involved the addition of message types that could be exchanged between the components, and the implementation of the message handling (and possibly responding) routines in the single components, as well as the network connection management and GUI rendering.

Overall, the implementation took about four full 12-hour-days, plus two additional ones for the update of the architectural models and the write-up of this document. Of these four implementation days, the first day succeeded in building up the structural component/class model and the composition of the single components and connectors. Thus, about 25% of the implementation time was spent on the mapping of the architectural model to the source code. Consequently, I can definitely see the advantage of using an architectural mapper tool that automatically generates this structure, especially if it can map changes in the sources back to the architectural models. I have finally acquired access to the 1.x-way mapper tool and am planning to look into how well the tool could be used to create an implementation class model similar to the one I manually implemented after the completion of the quarter (sadly, there currently are too many other projects that require immediate attention ..).

Another thought that occurred to me was the following: As it already seems to be possible to map the architectural models to source code automatically, is there really a need for an architecture description language that is separate from the implementation? It seems that the overall goal of the software architecture process is to maintain consistency between the models and the implementation code, so maybe it would be easier to create an implementation language that can explicitly encapsulate the architecture structure and models in itself (with programming constructs integrated into the language). I am not sure how such a language would explicitly look like, but it could be some kind of implementation language that is suitable for the architectural style followed (or domain-specific for the type application), or even an implementation language that is extensible so that new styles could be supported, and that is somehow annotated with additional information that is necessary for the architectural model. It would probably have to encapsulate the structural composition in a declarative way, while possibly adding declarative annotations to the behavior implementations (so that behavioral models could be expressed as well). I don't think this is an easy task, but it could be something for further investigation. Also, one would have to consider that the description of the architecture should be possible on a higher (more abstract) level without having to implement the detailed behavior first. So the programming language would have to support this as well. Such a language would solve the problem of architecture and implementation drifting away from each other, as the implementation defines the architecture itself. On the other hand, no mapping between architecture and implementation would be necessary anymore, as both are the same.

LESSONS LEARNED DURING THE IMPLEMENTATION

Something that I can take away from the implementation of the game in the C2 style is definitely that this kind of style is at least one level too complex for a simple game like this. Usually, a game like this could be implemented with roughly 10-20 classes, whereas the C2 implementation uses about 5-10 times as many, and probably similarly much more lines of code. Additionally, the imposed asynchrony constraints between components through the message exchange add a considerable amount of necessary attention to thread safety and can also complicate simple tasks as the fact that the order of received messages is uncertain has to be considered in every behavior implementation. The decoupling of state and logic components implies that each component has to hold the current game status (or at least, the part of the status that it requires to know) separately by evaluating the data store update notifications. Therefore, a lot of the game state is duplicated in the single components, thus increasing the risk for consistency problems (if there were small bugs in the components individual state holding functionality). Especially as it seems that the advantages gained through this decoupling of the C2 style (e.g. simple component exchange or addition, component reuse, different implementation languages for the single components, …) are not required for this project, it does not seem to be an ideal or even appropriate solution.

On the other hand, one advantage the C2 style gave was the easy design and implementation of the state transfer between clients and servers, as the C2 messages could just be used as state information carriers in this exchange.

Another slight confusion I have had from time to time, also during the architecture phase, was which of the messages actually notifications are and which requests. Sometimes, it is hard to distinguish between requests "to handle some information" and notifications about "new information that could be handled". The semantic difference between those two types of messages is that in the former case, as they are sent upwards in the C2 architecture, the origin component of the message is allowed to know about the services of the component that will handle the information, whereas in the latter case, it is not allowed to know about the services of the component that will/could handle the information. I for myself have a hard time understanding this difference, as I usually think of the components so (highly) decoupled from one another, that they neither know of components below nor above them. For example, it would have probably been more correct in the sense of the C2 style to place the Server/Client Message Composer components below the respective Connection Manager Components in the fine-grained structural architectural models of the Client to Server / Server to Clients connectors and have the components send handle requests instead of notifications about new messages ready to be handled / sent over the network as one could argue that the Composer components know of the service of sending messages. It is a tight call, and, for me, it is a place in the C2 style that leaves room for (mis)interpretation.

CHANGES APPLIED TO THE ARCHITECTURE MODELS

As described in the first section, several minor changes were made to the models. While parts of them resulted from minor fallacies during the architecture modeling (e.g. LevelStatus stored in Level Component instead of the Status Component, or Asteroid/Bullet destruction managing on the clients instead of the server), others resulted from the simple truth that the architecture phase just cannot be as detailed and deep-thought about each single component implementation as the implementation would require (for example, that in order for the clock ticks and thus the entity movements in the game to happen at the very same times on all clients, a time synchronization between clients and server, i.e. NTP connections, would be useful).

Generally speaking, the changes to the architecture models cannot be considered to be of major nature. I believe the reason for this (for why there were no big changes to be taken) lies in the preparatory research of game implementation techniques and of possible libraries to use (e.g. serialization/network libraries) I undertook before starting the modeling of the architecture. Thus, such a spike-like investigation of the application domain proved useful, especially for a developer who is otherwise rather inexperienced in the domain (like me in game programming …).

CONSISTENCY ARGUMENT

The consistency argument of the structural models was relatively easy to accomplish, because the structure is directly represented in the class / object model of the implementation. Proving that the behavioral models correspond to the implementation behavior was more difficult. For the GUI, I chose to explain the GUI implementation to prove its conformity, which was possible because of the small scope of the behavioral model (only represented behavior of the GUI component). While for the client initialization process model (sequence diagram), this approach could not be taken, as therefore a significant part of the implementation would have to have been explained in detail. Instead, I used the log of the message flow between components and connectors to show that the messages are exchanged in the same way as modeled in the diagram. This was only possible because of the abstract implementation of the C2Connectors and C2Components (as abstract Java classes), so they could easily be instrumented with this logging functionality at a central location in the application source code.

As the component/connector structure is explicitly modeled in the implementation, and no other ways for components to communicate with each other exist, and as the behavioral models of the GUI has been sufficiently proven to conform to the implementation, and as the (asynchronous) flow of messages represented in the sequence diagram of the client initialization is clearly visible in the same way in the logs, I am very confident that the architecture and the implementation are consistent.

VALUE OF THE ARCHITECTURAL MODELS

As I already expressed in my assessment of the architectural phase, I had serious doubts about the benefits of the detailed modeling of the game architecture in contrast to more adaptive / agile development methods (that promote the concepts of failing early and often, like Extreme Programming), at least for an application of a scope like this project. I for myself see the advantage of the explicit modeling of the architecture for this specific game in two aspects: Firstly, for academic learning purposes (see how software architecture is done), and secondly, to prevent mistakes in the implementation of the game. The former is definitely a valid point and I believe it is the reason for this exercise not being non-sense. ☺ In regards to the latter: I am talking about the prevention of faulty or not ideal design decisions. For example, in my modeling phase, I discovered that it would be useful to store the movement of ships, bullets and asteroids as position/speed/acceleration vectors that could be used to predict the movement of the ships on all clients without transferring every single position change over the network connection. Also, citing my assessment of the lessons learned about the system properties in the last document, I "could clarify the relationships between the individual components and the way information would get from the clients to the server and vice-versa" through modeling the client initialization behavior and that "modeling the client-server connector's inner structure showed that both a UDP and a TCP connection between each client and the server would be advisable". All these realizations were undoubtedly useful in the implementation phase, but do they really rectify the enormous effort to create those detailed architectural models?

If I hadn't made those observations, the problems they solve would have come up during the implementation phase. I would have either noticed them while implementing the respective parts of the system and thinking about the implications of my doings, or would have stumbled upon them once the effect of the problem would manifest itself in the game during testing. In the first case, I might have been able to prevent a mistake during the development without making it, while in the second, I would have made it and would have had to correct it. An example for the latter case would be using Cartesian coordinates for the representation of the acceleration and speed vectors. I only realized that this representation would not allow the direction of the vector to be retained if its magnitude was 0, thus for a ship that did not accelerate, its steering direction would be lost. I could correct this mistake easily, once I realized it, by making extensive use of Eclipse's refactoring support in a matter of a few minutes. Detecting this kind of problem and preventing it in the architecture phase would have required a much more detailed model of the steering process, and as such a model of every single components behavior would be a huge investment in the architecture phase, this would probably not be feasible.

So, the question is, for which of the design decisions is it actually feasible to create a detailed architectural model for in the first place? Which of the possible mistakes in the implementation I could have made if I did not have the insights of the architectural phase require less effort in the architectural phase to be become visible than they would require refactoring the implementation to correct the mistake?

Specifically for this project, my opinion is that none of these insights rectify the effort of the formal and detailed architectural models. I for myself believe that for a project like this, a few less detailed and informal models would have more than sufficed to detect the same problems and gain similar insights without putting a huge effort into the architectural phase. Also I believe that even if some insights would not be possible to gain through such a modeling, those would probably be minor and could easily be corrected during the implementation. Because of this, for the next project of similar scope, I would only take a path that includes a short, informal architectural phase.

Nevertheless, it should be noted that this assessment is very specific to the project undertaken. In other cases, more detailed models could make sense (for example, if they are used to convey information about design decisions to stake holders), or very formal models would be advantageous (e.g. for analytic processing, including model or constraint checking and validation of correctness). It is always necessary to evaluate whether the benefits gained through a specific model outweigh the effort it requires to create, individually for each single project.

APPENDIX

SOURCE CODE: BUILD UP CLIENTTOSERVERCONNECTOR INNER STRUCTURE

```
clientMessageComposerComponent = new ClientMessageComposerComponent();
serverMessageHandlerComponent = new ServerMessageHandlerComponent();
serverConnectionStateChangeHandlerComponent = new ServerConnectionStateChangeHandlerComponent();
serverConnectionManagerComponent = new ServerConnectionManagerComponent();

thresholdingPortConnector = new ThresholdingPortConnector();
dataStorePortConnector = new DataStorePortConnector();
connectionManagerConnector = new ConnectionManagerConnector();

thresholdingPortConnector
  .getPort(
    ThresholdingPortConnector.PORT_IDENTIFIER_CLIENT_MESSAGE_COMPOSER_COMPONENT)
  .connectTo(
    clientMessageComposerComponent
      .getPort(ClientMessageComposerComponent.PORT_IDENTIFIER_THRESHOLDING_PORT_CONNECTOR));

dataStorePortConnector
  .getPort(
    DataStorePortConnector.PORT_IDENTIFIER_SERVER_MESSAGE_HANDLER_COMPONENT)
  .connectTo(
    serverMessageHandlerComponent
      .getPort(ServerMessageHandlerComponent.PORT_IDENTIFIER_DATA_STORE_PORT_CONNECTOR));

dataStorePortConnector
  .getPort(
    DataStorePortConnector.PORT_IDENTIFIER_SERVER_CONNECTION_STATE_CHANGE_HANDLER_COMPONENT)
  .connectTo(
    serverConnectionStateChangeHandlerComponent
      .getPort(ServerConnectionStateChangeHandlerComponent.
                 PORT_IDENTIFIER_DATA_STORE_PORT_CONNECTOR));

connectionManagerConnector
  .getPort(
    ConnectionManagerConnector.PORT_IDENTIFIER_CLIENT_MESSAGE_COMPOSER_COMPONENT)
  .connectTo(
    clientMessageComposerComponent
      .getPort(ClientMessageComposerComponent.PORT_IDENTIFIER_CONNECTION_MANAGER_CONNECTOR));

connectionManagerConnector
  .getPort(
    ConnectionManagerConnector.PORT_IDENTIFIER_SERVER_MESSAGE_HANDLER_COMPONENT)
  .connectTo(
    serverMessageHandlerComponent
      .getPort(ServerMessageHandlerComponent.PORT_IDENTIFIER_CONNECTION_MANAGER_CONNECTOR));

connectionManagerConnector
  .getPort(
    ConnectionManagerConnector.PORT_IDENTIFIER_SERVER_CONNECTION_STATE_CHANGE_HANDLER_COMPONENT)
  .connectTo(
    serverConnectionStateChangeHandlerComponent
      .getPort(ServerConnectionStateChangeHandlerComponent.
                 PORT_IDENTIFIER_CONNECTION_MANAGER_CONNECTOR));

connectionManagerConnector
  .getPort(
    ConnectionManagerConnector.PORT_IDENTIFIER_SERVER_CONNECTION_MANAGER_COMPONENT)
  .connectTo(
    serverConnectionManagerComponent
      .getPort(ServerConnectionManagerComponent.PORT_IDENTIFIER_CONNECTION_MANAGER_CONNECTOR));
```

LOG CLIENT INITIALIZATON (CLIENT-SIDE)

```
2013-02-27 02:48:37,362 INFO  [main] Client: starting up ...
2013-02-27 02:48:37,628 INFO  [Thread-13] ServerConnectionManagerComponent: sending host discovery packet
2013-02-27 02:48:43,928 TRACE [pool-1-thread-1] ServerConnectionManagerComponent: new message: CurrentEntityDataNotification
2013-02-27 02:48:43,929 TRACE [pool-1-thread-1] ServerConnectionManagerComponent: new message: CurrentGameStartTimeNotification
2013-02-27 02:48:43,932 TRACE [pool-1-thread-1] ServerConnectionManagerComponent: new message: CurrentEntityDataNotification
2013-02-27 02:48:43,941 TRACE [Thread-19] ServerMessageHandlerComponent: sending request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,941 TRACE [Thread-19] DataStorePortConnector: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,942 TRACE [Thread-17] DataStorePortConnector: sending request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,943 TRACE [Thread-17] DataStoreConnector: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,943 TRACE [pool-1-thread-1] ServerConnectionManagerComponent: new message: CurrentEntityDataNotification
2013-02-27 02:48:43,943 TRACE [pool-1-thread-1] ServerConnectionManagerComponent: new message: CurrentEntityDataNotification
2013-02-27 02:48:43,944 TRACE [Thread-8] DataStoreConnector: sending request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,944 TRACE [Thread-8] AsteroidsComponent: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,944 TRACE [Thread-8] ClockComponent: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,944 TRACE [Thread-8] StatusComponent: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,945 TRACE [Thread-8] LevelComponent: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,945 TRACE [Thread-8] ShipsComponent: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,945 TRACE [Thread-8] BulletsComponent: receiving request: UpdateGameStartTimeRequest
2013-02-27 02:48:43,946 TRACE [pool-1-thread-1] ServerConnectionManagerComponent: new message: CurrentEntityDataNotification
2013-02-27 02:48:43,951 TRACE [Thread-19] ServerMessageHandlerComponent: sending request: UpdateLevelMapRequest
2013-02-27 02:48:43,951 TRACE [Thread-19] DataStorePortConnector: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,951 TRACE [Thread-19] ServerMessageHandlerComponent: sending request: UpdateLevelStatusRequest
2013-02-27 02:48:43,951 TRACE [Thread-19] DataStorePortConnector: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,952 TRACE [Thread-17] DataStorePortConnector: sending request: UpdateLevelMapRequest
2013-02-27 02:48:43,953 TRACE [Thread-17] DataStoreConnector: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,953 TRACE [Thread-17] DataStorePortConnector: sending request: UpdateLevelStatusRequest
2013-02-27 02:48:43,954 TRACE [Thread-8] DataStoreConnector: sending request: UpdateLevelMapRequest
2013-02-27 02:48:43,954 TRACE [Thread-8] AsteroidsComponent: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,954 TRACE [Thread-17] DataStoreConnector: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,955 TRACE [Thread-8] ClockComponent: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,955 TRACE [Thread-8] StatusComponent: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,956 TRACE [Thread-8] LevelComponent: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,956 TRACE [Thread-8] LevelComponent: sending notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,956 TRACE [Thread-8] DataStoreConnector: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,956 TRACE [Thread-8] LevelComponent: sending notification: LevelUpdatedNotification
2013-02-27 02:48:43,956 TRACE [Thread-8] DataStoreConnector: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,957 TRACE [Thread-7] DataStoreConnector: sending notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,957 TRACE [Thread-8] ShipsComponent: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,958 TRACE [Thread-7] CollisionDetectionLogicComponent: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,958 TRACE [Thread-8] BulletsComponent: receiving request: UpdateLevelMapRequest
2013-02-27 02:48:43,958 TRACE [Thread-8] DataStoreConnector: sending request: UpdateLevelStatusRequest
2013-02-27 02:48:43,958 TRACE [Thread-7] ThresholdingLogicComponent: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,958 TRACE [Thread-8] AsteroidsComponent: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,959 TRACE [Thread-8] ClockComponent: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,959 TRACE [Thread-7] LogicConnector: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,959 TRACE [Thread-8] StatusComponent: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,959 TRACE [Thread-7] GUIComponent: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,959 TRACE [Thread-8] LevelComponent: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,960 TRACE [Thread-9] LogicConnector: sending notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,960 TRACE [Thread-8] ShipsComponent: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,960 TRACE [Thread-9] GameLogicComponent: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,960 TRACE [Thread-8] BulletsComponent: receiving request: UpdateLevelStatusRequest
2013-02-27 02:48:43,961 TRACE [Thread-9] GameLogicComponent: sending request: CreateNewShipRequest
2013-02-27 02:48:43,961 TRACE [Thread-9] LogicConnector: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,961 TRACE [Thread-7] PositionLogicComponent: receiving notification: LevelStatusUpdatedNotification
2013-02-27 02:48:43,961 TRACE [Thread-7] DataStoreConnector: sending notification: LevelUpdatedNotification
2013-02-27 02:48:43,962 TRACE [Thread-10] LogicConnector: sending request: CreateNewShipRequest
2013-02-27 02:48:43,962 TRACE [Thread-7] CollisionDetectionLogicComponent: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,963 TRACE [Thread-7] ThresholdingLogicComponent: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,965 TRACE [Thread-7] LogicConnector: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,965 TRACE [Thread-7] GUIComponent: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,966 TRACE [Thread-10] DataStoreConnector: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,966 TRACE [Thread-8] DataStoreConnector: sending request: CreateNewShipRequest
2013-02-27 02:48:43,966 TRACE [Thread-8] AsteroidsComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,967 TRACE [Thread-8] ClockComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,968 TRACE [Thread-8] StatusComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,968 TRACE [Thread-9] LogicConnector: sending notification: LevelUpdatedNotification
2013-02-27 02:48:43,969 TRACE [Thread-8] LevelComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,969 TRACE [Thread-8] ShipsComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,969 TRACE [Thread-9] GameLogicComponent: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,969 TRACE [Thread-8] ShipsComponent: sending notification: NewShipNotification
2013-02-27 02:48:43,969 TRACE [Thread-8] DataStoreConnector: receiving notification: NewShipNotification
2013-02-27 02:48:43,970 TRACE [Thread-8] BulletsComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:43,996 TRACE [Thread-7] PositionLogicComponent: receiving notification: LevelUpdatedNotification
2013-02-27 02:48:43,997 TRACE [Thread-7] DataStoreConnector: sending notification: NewShipNotification
2013-02-27 02:48:43,998 TRACE [Thread-7] CollisionDetectionLogicComponent: receiving notification: NewShipNotification
2013-02-27 02:48:43,998 TRACE [Thread-7] ThresholdingLogicComponent: receiving notification: NewShipNotification
2013-02-27 02:48:43,998 TRACE [Thread-7] ThresholdingLogicComponent: sending notification: NewShipNotification
2013-02-27 02:48:43,999 TRACE [Thread-7] ThresholdingPortConnector: receiving notification: NewShipNotification
2013-02-27 02:48:44,000 TRACE [Thread-14] ThresholdingPortConnector: sending notification: NewShipNotification
2013-02-27 02:48:44,000 TRACE [Thread-7] LogicConnector: receiving notification: NewShipNotification
2013-02-27 02:48:44,000 TRACE [Thread-14] ClientMessageComposerComponent: receiving notification: NewShipNotification
2013-02-27 02:48:44,001 TRACE [Thread-9] LogicConnector: sending notification: NewShipNotification
2013-02-27 02:48:44,001 TRACE [Thread-7] GUIComponent: receiving notification: NewShipNotification
2013-02-27 02:48:44,002 TRACE [Thread-9] GameLogicComponent: receiving notification: NewShipNotification
2013-02-27 02:48:44,002 TRACE [Thread-18] ServerConnectionManagerComponent: sending message to server (TCP): NewShipNotification
2013-02-27 02:48:44,004 TRACE [Thread-7] PositionLogicComponent: receiving notification: NewShipNotification
```

LOG OF CLIENT INITIALIZATON (SERVER-SIDE)

```
2013-02-27 02:48:43,895 INFO  [pool-1-thread-1] ClientConnectionManagerComponent: client connected: 1
2013-02-27 02:48:43,900 TRACE [Thread-12] ClientConnectionStateChangeHandlerComponent: sending request: GetCurrentDataRequest
2013-02-27 02:48:43,900 TRACE [Thread-12] DataStorePortConnector: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,903 TRACE [Thread-10] DataStorePortConnector: sending request: GetCurrentDataRequest
2013-02-27 02:48:43,905 TRACE [Thread-10] DataStoreConnector: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,905 TRACE [Thread-10] ClientStatesComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,905 TRACE [Thread-5] DataStoreConnector: sending request: GetCurrentDataRequest
2013-02-27 02:48:43,906 TRACE [Thread-5] AsteroidsComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,907 TRACE [Thread-5] AsteroidsComponent: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,907 TRACE [Thread-5] DataStoreConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,907 TRACE [Thread-5] ClockComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,908 TRACE [Thread-5] ClockComponent: sending notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,908 TRACE [Thread-5] DataStoreConnector: receiving notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,908 TRACE [Thread-5] StatusComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,908 TRACE [Thread-5] StatusComponent: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,909 TRACE [Thread-5] DataStoreConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,909 TRACE [Thread-5] LevelComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,909 TRACE [Thread-5] LevelComponent: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,909 TRACE [Thread-5] DataStoreConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,910 TRACE [Thread-5] ShipsComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,910 TRACE [Thread-5] ShipsComponent: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,910 TRACE [Thread-4] DataStoreConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,910 TRACE [Thread-5] DataStoreConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,911 TRACE [Thread-5] BulletsComponent: receiving request: GetCurrentDataRequest
2013-02-27 02:48:43,911 TRACE [Thread-5] BulletsComponent: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,911 TRACE [Thread-5] DataStoreConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,915 TRACE [Thread-4] LevelManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,916 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,917 TRACE [Thread-4] DataStorePortConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,918 TRACE [Thread-4] DataStoreConnector: sending notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,919 TRACE [Thread-9] DataStorePortConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,920 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,921 TRACE [Thread-4] LevelManagerComponent: receiving notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,921 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification:
CurrentEntityDataNotification
2013-02-27 02:48:43,922 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,923 TRACE [Thread-4] DataStorePortConnector: receiving notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,924 TRACE [Thread-4] DataStoreConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,925 TRACE [Thread-9] DataStorePortConnector: sending notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,925 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: CurrentGameStartTimeNotification
2013-02-27 02:48:43,925 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to client 1 (TCP):
CurrentEntityDataNotification
2013-02-27 02:48:43,926 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification:
CurrentGameStartTimeNotification
2013-02-27 02:48:43,926 TRACE [Thread-4] LevelManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,928 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to client 1 (TCP):
CurrentGameStartTimeNotification
2013-02-27 02:48:43,928 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,928 TRACE [Thread-4] DataStorePortConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,929 TRACE [Thread-9] DataStorePortConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,929 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,929 TRACE [Thread-4] DataStoreConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,931 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification:
CurrentEntityDataNotification
2013-02-27 02:48:43,931 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to client 1 (TCP):
CurrentEntityDataNotification
2013-02-27 02:48:43,931 TRACE [Thread-4] LevelManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,933 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,934 TRACE [Thread-4] DataStorePortConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,935 TRACE [Thread-9] DataStorePortConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,935 TRACE [Thread-4] DataStoreConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,936 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,936 TRACE [Thread-4] LevelManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,937 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,937 TRACE [Thread-4] DataStorePortConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,938 TRACE [Thread-4] DataStoreConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,938 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification:
CurrentEntityDataNotification
2013-02-27 02:48:43,938 TRACE [Thread-9] DataStorePortConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,938 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,938 TRACE [Thread-4] LevelManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,940 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,940 TRACE [Thread-4] DataStorePortConnector: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,941 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to client 1 (TCP):
CurrentEntityDataNotification
2013-02-27 02:48:43,941 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification:
CurrentEntityDataNotification
2013-02-27 02:48:43,943 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to client 1 (TCP):
CurrentEntityDataNotification
2013-02-27 02:48:43,941 TRACE [Thread-9] DataStorePortConnector: sending notification: CurrentEntityDataNotification
2013-02-27 02:48:43,944 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: CurrentEntityDataNotification
2013-02-27 02:48:43,944 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification:
CurrentEntityDataNotification
2013-02-27 02:48:43,945 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to client 1 (TCP):
CurrentEntityDataNotification
2013-02-27 02:48:44,007 TRACE [pool-1-thread-1] ClientConnectionManagerComponent: new message from client 1: NewShipNotification
2013-02-27 02:48:44,012 TRACE [Thread-12] ClientMessageHandlerComponent: sending request: CreateNewShipRequest
2013-02-27 02:48:44,012 TRACE [Thread-12] DataStorePortConnector: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,014 TRACE [Thread-10] DataStorePortConnector: sending request: CreateNewShipRequest
2013-02-27 02:48:44,015 TRACE [Thread-10] DataStoreConnector: receiving request: CreateNewShipRequest
```

```
2013-02-27 02:48:44,016 TRACE [Thread-5] DataStoreConnector: sending request: CreateNewShipRequest
2013-02-27 02:48:44,016 TRACE [Thread-10] ClientStatesComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,017 TRACE [Thread-5] AsteroidsComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,017 TRACE [Thread-5] ClockComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,018 TRACE [Thread-5] StatusComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,019 TRACE [Thread-5] LevelComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,019 TRACE [Thread-5] ShipsComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,020 TRACE [Thread-5] ShipsComponent: sending notification: NewShipNotification
2013-02-27 02:48:44,020 TRACE [Thread-5] DataStoreConnector: receiving notification: NewShipNotification
2013-02-27 02:48:44,020 TRACE [Thread-5] BulletsComponent: receiving request: CreateNewShipRequest
2013-02-27 02:48:44,021 TRACE [Thread-4] DataStoreConnector: sending notification: NewShipNotification
2013-02-27 02:48:44,023 TRACE [Thread-4] LevelManagerComponent: receiving notification: NewShipNotification
2013-02-27 02:48:44,025 TRACE [Thread-4] AsteroidManagerComponent: receiving notification: NewShipNotification
2013-02-27 02:48:44,026 TRACE [Thread-4] DataStorePortConnector: receiving notification: NewShipNotification
2013-02-27 02:48:44,027 TRACE [Thread-9] DataStorePortConnector: sending notification: NewShipNotification
2013-02-27 02:48:44,027 TRACE [Thread-9] ServerMessageComposerComponent: receiving notification: NewShipNotification
2013-02-27 02:48:44,029 TRACE [Thread-11] ClientConnectionManagerComponent: sending message to all clients except 1 (TCP):
NewShipNotification
2013-02-27 02:48:44,030 TRACE [Thread-9] ClientConnectionStateChangeHandlerComponent: receiving notification: NewShipNotification
```

REFERENCES

[1] Atari “Lunar Lander” online arcarde game. http://atari.com/arcade#!/arcade/lunarlander/

[2] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. 2009. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.

[3] Kryonet - network and serialization library for Java. https://code.google.com/p/kryonet/

[4] Abstract Windowing Toolkit. http://docs.oracle.com/javase/7/docs/technotes/guides/awt/index.html

[5] Apache Commons Net. http://commons.apache.org/proper/commons-net/

[6] Yongjie Zheng. 2011. 1.x-Way architecture-implementation mapping. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA.