Table of Contents

1. Game description

The game is an abstracted real time strategy (RTS) game in which the players manage the ships between different planets. This game takes heavy influence from the game Galcon. Each planet continuously generates more ships for their owner.

When the game starts, all players control one planet from the selection of the planets of the map and expand outwards conquering other planets from a neutral faction or other players. The goal of the game is to annihilate the opposing players by capturing their planets.

The gameplay simply consists of directing the troops to capture or defend the best possible objectives for each moment. Not all ships on a planet are required to be sent, an user selectable percentage of ships can be sent.

2. Game mechanics and rules

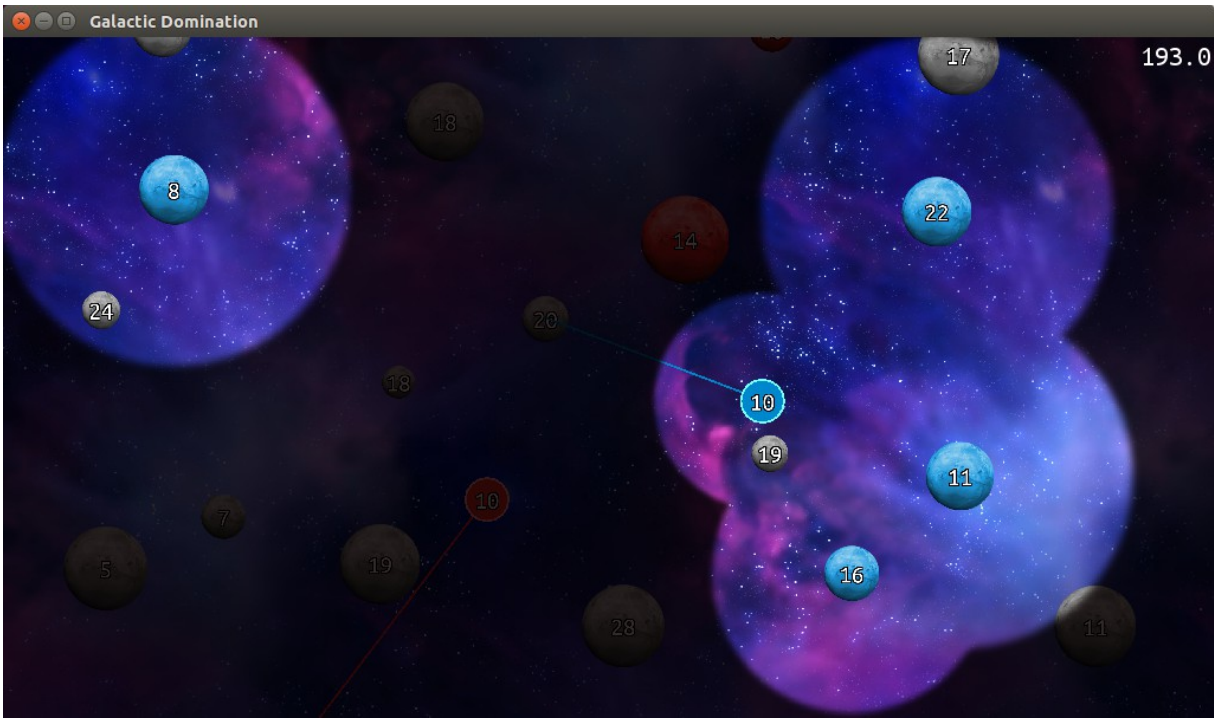
- The player starts with at least one planet
- All the planets that are player controlled generate units over time
- These units can be used to send attacks:
    - The player selects the source and the destination planet and the percentage of units to send
    - The sent units move through the space at fixed speed
    - When the units reach the destination planet
        - A) If the planet is friendly, the units are added up to the amount currently stationed on the planet
        - B) If the planet is hostile (or neutral), an instant battle takes place where a difference of the units is calculated, and the control of the planet is given to the player who had the greater amount of units at the end of the battle.
- The game uses a fog of war system, where:
    - Vision is generated around friendly planets and flying battle fleets
    - The original idea is that you only see the owners of the planets inside the fog, but nothing else
    - The player can see the number of stationed units at the planets only if it is inside friendly visibility range
    - Also, you can see the target of enemy battle fleets when they are inside your vision range
- If a player leaves the game while its active, all his planets and fleets turn under neutral control
- The game ends when there is only one player remaining with controlled planets or fleets

2.1 Gameplay

Each player has an unique color, which is used to identify their planets and fleets. All the neutral planets are gray in color.

In the beginning of the game the view is centered to the planet(s) the player initially controls. The view can be moved by moving mouse to the edges of the screen. Optionally the player can zoom the game area in and out to see a smaller/greater view of the war zone at once using the mouse wheel.

To send fleets to other planets, the player selects a planet with the mouse cursor and drags to the destination planet. The selected planet and the target planet are indicated during this command. After the mouse button is released, the fleet is sent to the destination planet. The amount (percentage) of units sent can be controlled. The movement of these fleets and the generation of new units can be seen in real time.



A screenshot from a demo build

3. Game design

3.1 Basic Features

The game consists of a dedicated server and client applications. It can be played through networks that support UDP over (IPv4)/IPv6.

The game has a lobby system, where players must join a lobby before starting a game. After a game has started, no new players can join the game before it has ended. This ending condition is explained in the *game mechanics and rules* –section.

The servers can report to a master server that they are willing to take players in. This master server generates a list of all available servers, that the clients can request using UDP. The return packet contains the server names, IP-addresses and how many players there were on the server during the last received announce packet from the server.

The game has a chat that can be used to send messages to all players, or a specific player (whisper system). This chat also uses UDP-protocol.

Latency and jitter are measured with dedicated packets that are used to measure of the quality of the connection.

The game will be tested by at least by three non-group members. Their findings will be reported and included in the final documentation.

3.2 Implementation details

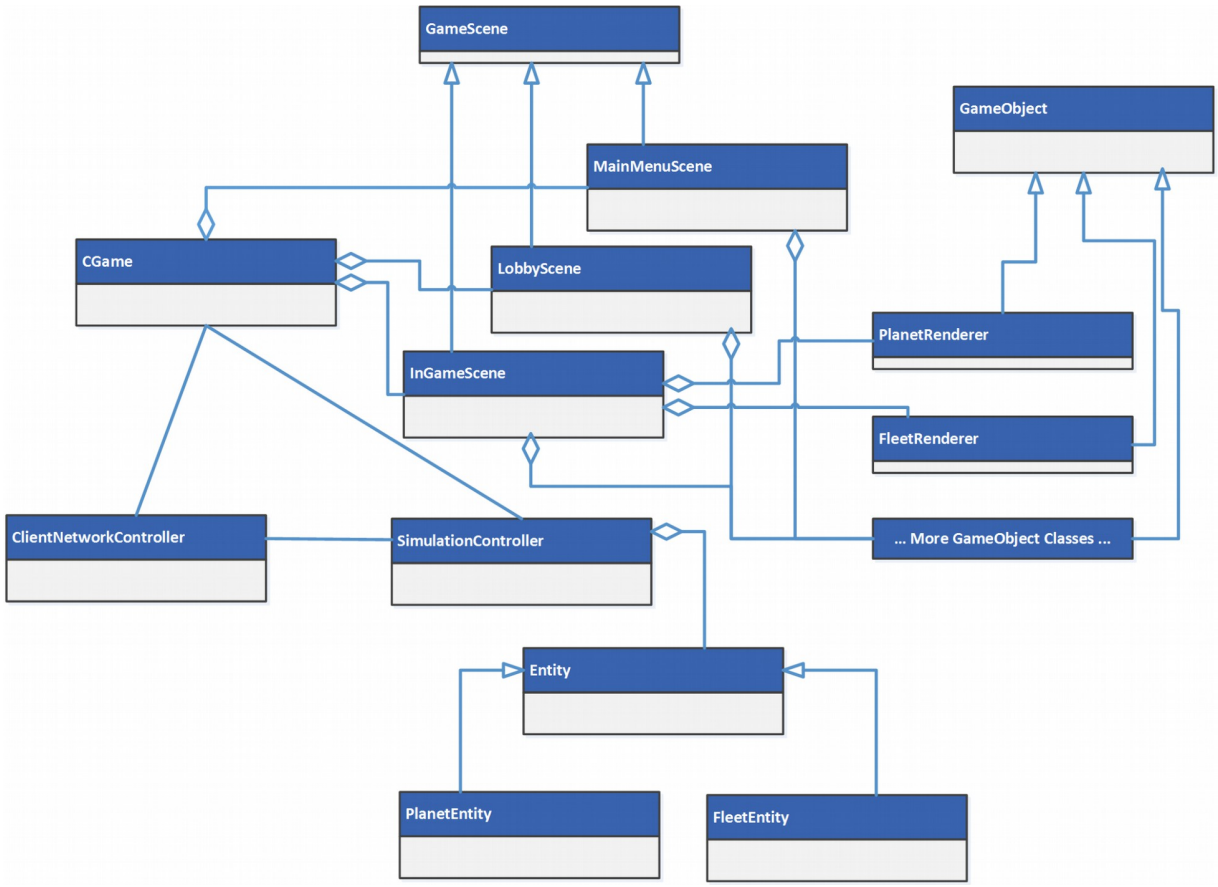
3.2.1 Client

The game uses Simple and Fast Multimedia Library (SFML) for the graphical representation of the game at the client side. A part of the challenge with a graphical user interface was that a simple system to handle the representation of the different objects, and the input events such as mouse and keyboard presses in the game had to be put in place.

The client / the UI is based on a *CGame* class, which represents a connection point for all the necessary parts for the game to work. The *CGame* runs a single *GameScene* at a time. There are different kinds of *Scenes* for different parts of the game, such as the different views in the menus, and the gameplay phase itself. These *Scenes* consist of *GameObjects* which have of the main logic for handling the event and rendering calls that are passed from the *CGame* to the *Scenes* all the way down to the *GameObjects*.

For the non-networked parts of the game these *GameObjects* work autonomously. But for the networked parts of the game, these *GameObjects* receive information from the *NetworkController* and the *SimulationController*, which is passed via the *CGame* class to matching handler calls. For example, this passed information populates the server listings in the server search scene, or the player information in the lobby scene.

Especially interesting is how the gameplay part itself is structured to work. The *SimulationController* is in charge to interpret the gameplay state information received from the server, which is passed to it by the *ClientNetworkController.* Since the server uses an *Entity*-based system for the internal presentation of the simulation's instance data, this is what is used on the clients' side as well. However, on the client's side each of these entities is also linked into a *PlanetRenderer* or *FleetRenderer GameObject* which are used to represent visually the data received from the server.



A simplified class diagram representing the client's structure

This distinction between the *Entities* and *GameObjects* is also important, since the actual game is based on the quite slowly updating tick based system. Even though the game is a real time strategy game, it is not especially very fast paced one. In the current implementation the actual logic side of the game is updated at 250 millisecond intervals, or at 4Hz. In our prototypes this proved to be more than enough for a smooth gameplay experience. This tickrate could also be scaled to adjust to the current connection conditions, meaning that if all the players have a good connection to the server, then the tickrate could be increased, and vice versa.

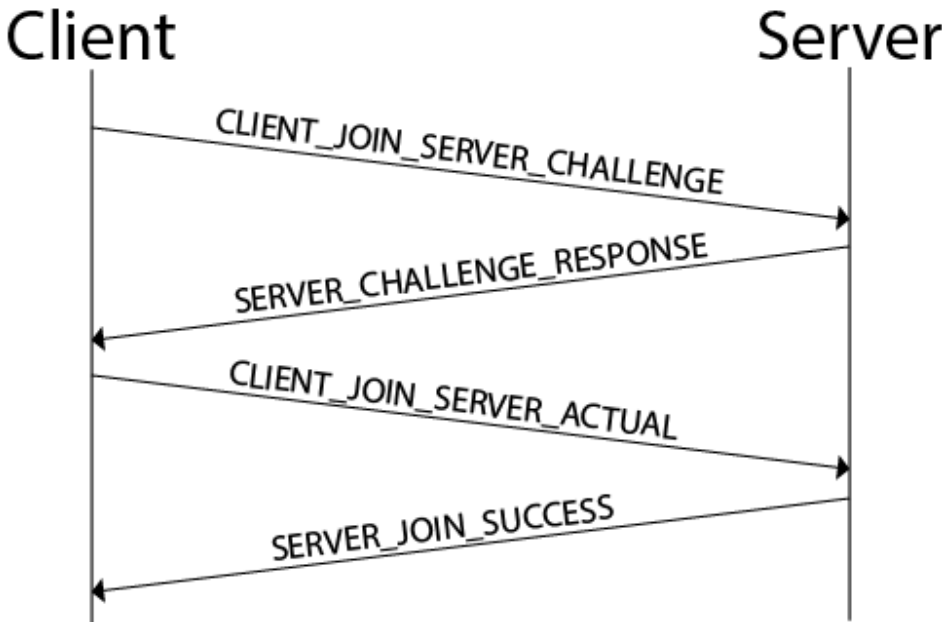
The logical representation of the game is updated relatively slowly, and if it would be displayed straight out as it is, the end result would be very skippy. An interpolation scheme ensures that client experience is still smooth even though the actual data is updated at a much slower rate than what the client's refresh rate is. This is where the distinction of the *Entities* and the

GameObjects comes into play on the client's side. As the tick length is strictly determined, all the entity updates can be gradually applied to the graphical representations (*GameObjects)* of the *Entities* using the tick length and a delta timing scheme.

3.2.2 Server

The server follows the same principles as the client does. It has a separate handler for the network connections, and then it has its own controller for the simulation flow. The network data gets passed and parsed into simulation actions, which then, when processed, are sent back to the clients as entity updates to be executed. In practice this is achieved by polling the server socket for incoming messages, and during the timeout periods of the poller to check if there is timed events that need to be processed.

Probably the most interesting aspect of the server is how the clients connect to it. A challenge system is in place. This is because since the game uses UDP for its connections, which means that the connection packets would be easy to spoof, as there is no handshake protocol built into UDP. To actually connect as a player to the game server, the client has to reply with a challenge key sent by the server. This also works as a handshake for the reliability layer, which will be talked more in detail later on.



Client to server connection handshake message sequence chart

3.2.3 Master Server

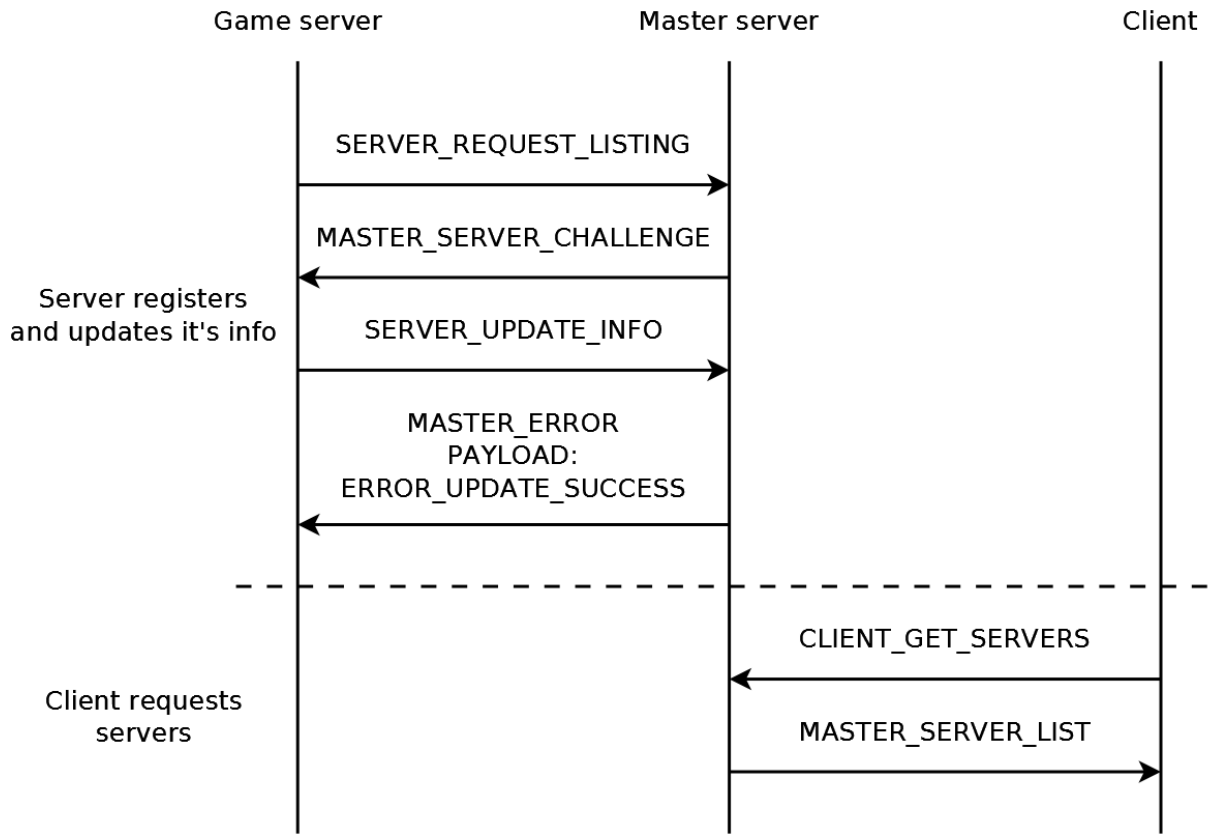
Clients need to be able to find servers to play on and one of the easiest ways to provide this information is to use centralized “Master server”. Thus the purpose of the master server is to provide interface for game servers to get registered and return this information to the clients on request.

Implementation of the master server can be broken into two distinct parts: handling of the network packets and handling server listing. Packet handling uses the common network classes such as *Socket* and *NetworkPacket* to receive and handle packets.

Servers are stored in map data container using the ip address and port number as a key. This way servers can be found fast when they update their information. This means that the map need custom comparison object to sort the servers correctly. When new server wants to be listed it is first added as temporary server. This means that it will not be send to clients when client requests servers because not all of the information is available. Temporary servers are automatically converted to “real” servers when the server updates its information. Master server returns a random number as a confirmation challenge that the server was added as a temporary server. This challenge must be used to by the server to update its information. This works as a safeguard against packet spoofing.

Master server also incorporates a timeout feature that purges the server list so that servers that haven’t updated their information are removed. This feature uses separate list data structure that contains iterators to the serverdata in the map data structure. Because temporary servers have different timeout period than the actual registered servers this list is divided into two halves. These halves are separated using “m_timeoutAnchor” iterator according to following rule. Servers from list.begin() → m_timeoutAnchor are temporary servers so that server nearest to the list.begin are going to timeout first. And m_timeoutAnchor → list.end() are fully registered servers and they are organized in similar manner: servers nearest to the m_timeoutAnchor are going to timeout first. When server is added as temporary server it is inserted left of the m_timeoutAnchor (will timeout last). When server updates its information it is removed from wherever it originally was and is reinserted to the end of the list. Usage of only one list was conscious decision because this way there is no need to know what state the server is in when it is first removed to be reinserted. If two different lists were used it would be mandatory to first

know if the server was temporary or fully registered server because it would be stored in different list depending on this fact.



Message sequence chart of connections made to the master server

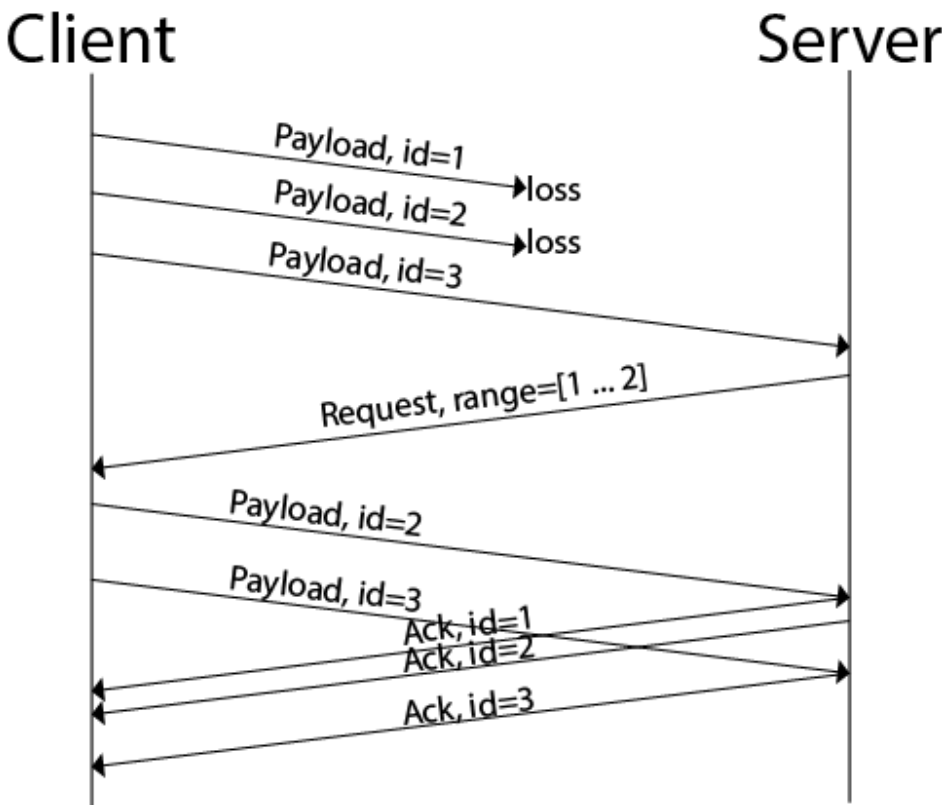
3.2.4 Reliability layer

After the initial connection establishment, all the game packets go through a custom reliability layer. This layer makes sure that all the packets are processed in-order and if a packet goes missing it can be resent. Both the clients and the server use the same general implementation of the protocol.

In more detail: first a payload packet is sent and saved to a temporary re-sending buffer. The packet contains a running sequence number used to both distinguish the packets and to make sure they are in order.

When the peer receives the packet it checks if it's number is as expected and sends an acknowledgement packet containing the sequence number of the received packet. When this acknowledgement is received by the original sender, all packets equal to or less the packet number are erased from the re-sending buffer and a counter is updated to mark the sequence number of this latest ack'd packet.

If the received packet has a number lower than expected it is silently ignored as it is an unnecessary duplicate. However, if the number is higher, the packet is saved to a buffer to wait for it's turn. No ack is sent yet, only after it is the packet's turn.



Payload message flow sequence chart (roles interchangeable)

Additionally as each packet requires an acknowledgement, the ack packets are timed. This information can then be used to measure the ping and jitter of the connection. The health of the connection is further tracked with timers.

The input timer is reset with incoming handshake replies and when a payload packet in correct order is received. When the client detects that no packet has been received in 8 seconds it gives an error message and closes the connection. Additionally if the timer crosses a threshold(ex. two seconds), a special null-payload is sent as a keepalive. As this message also carries the sequence number it will cause the peer's reliability stack to either send an acknowledgement or a request for missing packets. Additionally the peer could then reset it's output timer, as the connection is ok.

The output timer is reset when an ack is received. For the connections the game is meant to be played, the input timer should be enough to keep the connection alive. If greater durability for packet loss is desirable the output timer could be monitored and the unconfirmed packets resent it the timer greatly crosses over the monitored ping and jitter average. Eventually an ACK arrives or the input timer expires and the connection is closed.

3.2.4.1 Security considerations

The current implementation assumes no malice. That is, buffer space is unlimited. A nasty peer could run the game normally, but also send a constant stream of packets with a very high sequence number, eventually filling up all the available memory of the receiver. To combat this a moderately sized limit could be implemented and a connection could then be terminated if too many packets are in the buffer.

Also there is no rate limiting of any kind. A bad client could send a large amount of chat messages for example and flood the connection of both the server and the connected clients.

3.2.4.2 Other

As there is no rate limiting, a very large amount of data can lead to suboptimal performance as packets keep getting lost in retransmission floods. The resending algorithm is also quite naïve. In situations where multiple out-of-order packets are received each of them generates a resend request for the same packets, and the peer then re-sends the packets that many times even if the requests arrive at the same time. If the packet loss was due to congestion this will just make the situation worse.

4. Protocol definitions

Derived units	Size in bytes	Description
string	uint16 + length of str bytes (2 + n)	A string which isn't null terminated, but is determined by the size in the 2 first bytes. Limited in length to STRING_BUFFER_SIZE.
bool	uint8 (1)	0 for false. All else considered as true.
float	int32 (4)	float value multiplied/divided by a fixed factor of 10 000 giving 4 decimal places of precision which should be enough for our purposes.

4.1 Reliability layer connection establishment

From:	Client
To:	Server
Packet type:	Join Server Challenge
Enum Identifier:	CLIENT_JOIN_SERVER_CHALLENGE
Description	1st step towards connecting to the server
Payload	uint8 - PACKET ID
Misc	Part of the reliability layer handshake

From:	Server
To:	Client
Packet type:	Server Challenge Response
Enum Identifier:	SERVER_CHALLENGE_RESPONSE
Description	Response to the challenge sent by a client, contains a challenge key the client has to send back to make the connection to the server
Payload	uint8 - PACKET ID string - challenge key
Misc	Part of the reliability layer handshake

From:	Client
To:	Server
Packet type:	Client Join Server Actual
Enum Identifier:	CLIENT_JOIN_SERVER_ACTUAL
Description	Sending the challenge key received from the server, along with the username we want to use
Payload	uint8 - PACKET ID string - challenge key string - user name
Misc	Part of the reliability layer handshake

From:	Server
To:	Client
Packet type:	Server Join Success
Enum Identifier:	SERVER_JOIN_SUCCESS
Description	Response to a *Join Actual.* A new player id is sent to the client.
Payload	uint8 - PACKET ID uint8 - user ID
Misc	Part of the reliability layer handshake

From:	Server
To:	Client
Packet type:	Connection Error Message
Enum identifier:	SERVER_CONNECTION_ERROR_MESSAGE
Description	Sent upon an error situation, such as a failure to connect
Payload	uint8 - PACKET ID uint8 - error code
Misc	Part of the reliability layer handshake

4.2 Reliability layer details

All the reliability layer packets are bidirectional meaning that they can be sent and received by both the server and the client.

Packet type:	**Reliability layer payload packet**
Enum Identifier:	**COMMON_PAYLOAD**
Description	**Contains an embedded game message packet and a running sequence number**
Payload	**uint8 - PACKET ID** **uint32 - PACKET SEQUENCE NUMBER** **char* - Game message**

Packet type:	**Reliability layer unreliable payload packet**
Enum Identifier:	**COMMON_PAYLOAD_UNRELIABLE**
Description	**As all traffic passes through reliability layer, a special packet is reserved for payloads that do not actually need reliability(eg. relayed ping statistics)**
Payload	**uint8 - PACKET ID** **char* - Game message**

Packet type:	**Reliability layer resend request**
Enum Identifier:	**COMMON_RESEND_REQUEST**
Description	**Sent when a peer receives a packet with a seq number higher than expected**
Payload	**uint8 - PACKET ID** **uint32 - LAST ACKED SEQ NUMBER** **uint32 - LAST MISSED SEQ NUMBER**

Packet type:	**Reliability layer packet ack**
Enum Identifier:	**COMMON_ACK**
Description	**Sent to indicate that all packets less or equal to the seq number have been received. Sent after a packet has been processed in-order.**
Payload	**uint8 - PACKET ID** **uint32 - SEQ NUMBER**

Packet type:	**Reliability layer termination packet**
Enum Identifier:	**COMMON_DISCONNECT**
Description	**Can be sent to indicate that the peer quit and no longer wants any packets**
Payload	**uint8 - PACKET ID**

4.3 Game messages

From:	**Server**
To:	**Client**
Packet type:	**Error Message**
Enum Identifier:	**SERVER_ERROR_MESSAGE**
Description	**Sent upon an error situation.**
Payload	**uint8 - PACKET ID** **uint8 - error code**

From:	Server
To:	Client
Packet type:	Server Player Information
Enum Identifier:	SERVER_PLAYER_INFORMATION
Description	Sent to add new data to the client's "database" of players
Payload	uint8 - PACKET ID uint8 - number of players in this message --- list --- uint8 - player id string - user name bool/uint8 - ready state

From:	Client
To:	Server
Packet type:	Client My Ready State
Enum Identifier:	CLIENT_MY_READY_STATE
Description	Sent when the client decides to change his/her ready state in the lobby.
Payload	uint8 - PACKET ID bool/uint8 - new ready state

From:	Server
To:	Client
Packet type:	Server Player Ready State Change
Enum Identifier:	SERVER_CLIENT_READY_STATE_CHANGE
Description	Relays the infromation to all the clients when a player requests a ready state change.
Payload	uint8 - PACKET ID uint8 - player id bool/uint8 - new ready state

From:	Client
To:	Server
Packet type:	Client Graceful Disconnect
Enum Identifier:	CLIENT_GRACEFUL_DISCONNECT
Description	Sent when the client is about to leave the server.
Payload	uint8 - PACKET ID

From:	Server
To:	Client
Packet type:	Server Player Disconnected
Enum Identifier:	SERVER_PLAYER_DISCONNECT
Description	Reports when a player has left the server, so the clients can update their player databases, and optionally show a message that a player has left the game.
Payload	uint8 - PACKET ID uint8 - player id uint8 - reason for disconnect

From:	Server
To:	Client
Packet type:	Server: Game is starting
Enum Identifier:	SERVER_GAME_IS_STARTING
Description	The server reports that all the players (>= 2) have readied up and that the game is starting. The client moves to the in-game scene at this point.
Payload	uint8 - PACKET ID uint16 - (initial) milliseconds per tick used

From:	Server
To:	Client
Packet type:	Server: Entity Initialization
Enum Identifier:	SERVER_ENTITY_INIT
Description	A packet describing the actions during the given tick ID. Entity init packet represents all the new entities that should be placed into the game during this tick.
Payload	uint8 - PACKET ID uint32 - tick number in which this happens uint16 - number of entities in this packet --- list --- uint16 - entity id uint8 - entity type < entity data (depends on entity) >

From:	Server
To:	Client
Packet type:	Server: Entity Update
Enum Identifier:	SERVER_ENTITY_UPDATE
Description	A packet describing the actions during the given tick ID. Entity update packet represents old entities that have updated data.
Payload	uint8 - PACKET ID uint32 - tick number in which this happens uint16 - number of entities in this packet --- list --- uint16 - entity id uint8 - entity type uint8 - changed fields bitfield < entity data (depends on entity) >

From:	Server
To:	Client
Packet type:	Server: Entity Deletion
Enum Identifier:	SERVER_ENTITY_DELETE
Description	A packet describing the actions during the given tick ID. Entity deletion packet represents old entities that have reached the end of their lifespan and should be deleted during this tick.
Payload	uint8 - PACKET ID uint32 - tick number in which this happens uint16 - number of entities in this packet --- list --- uint16 - entity id

From:	Client
To:	Server
Packet type:	Client: Command Input
Enum Identifier:	CLIENT_INPUT_COMMANDS
Description	Player sends the list of commands that he/she would like to get executed. At the moment all the commands are just sending units from one entity to another.
Payload	uint8 - PACKET ID uint32 - tick number in which this happens uint8 - number of commands in this packet --- list --- uint16 - entity id from uint16 - entity id to uint8 - percentage of units (0-100)

From:	Client
To:	Server
Packet type:	Client: Tick finalized
Enum Identifier:	CLIENT_TICK_FINALIZED
Description	Sent when the player is ready to move to a next tick. This means that all the client has had enough time to simulate the simulation forward and that all the commands for future ticks have been inputted in earlier messages.
Payload	uint8 - PACKET ID uint32 - tick number in which this client is ready to move forwards to

From:	Server
To:	Client
Packet type:	Server: Tick finalized
Enum Identifier:	SERVER_TICK_FINALIZED
Description	A packet describing that all the ENTITY_ messages for this tick have now been sent, and that the tick data is complete and the tick is ready to be stepped into. I.e. the server gives the clients permission to simulate further.
Payload	uint8 - PACKET ID uint32 - tick number

From:	Client
To:	Server
Packet type:	Client: Send chat message
Enum Identifier:	CLIENT_SEND_CHAT
Description	Sent when the player wants to send a chat message to the other players. Both the private chat messages and global chat messages share the same message ID.
Payload	uint8 - PACKET ID uint8 - player ID to, 255 for global chat string - chat message

From:	Server
To:	Client
Packet type:	Server: Relay chat message
Enum Identifier:	SERVER_RELAY_CHAT
Description	Sent to relay a chat message received from a client. If this is a private message, the message is only relayed to the recipient.
Payload	uint8 - PACKET ID uint8 - player ID from uint8/bool - is private message? string- message

From:	Server
To:	Client
Packet type:	Server: Broadcast winner
Enum Identifier:	SERVER_WINNER_ID
Description	Sent when the game has been won by one of the players, i.e. only one player only has units on the field (neutral units do not count).
Payload	uint8 - PACKET ID uint8 - player ID won

From:	Server
To:	Client
Packet type:	Server: Game Over
Enum Identifier:	SERVER_GAME_OVER
Description	Sent some time after the SERVER_WINNER_ID. This packet marks the end of the game, and means that it is time to move back to the lobby state.
Payload	uint8 - PACKET ID

From:	Server
To:	Client
Packet type:	Server: Ping information
Enum Identifier:	SERVER_RELAY_PING
Description	Sent periodically with unreliable packets by the server to notify all clients of the current ping and jitter timings.
Payload	uint8 - PACKET ID uint8 - number of players in this message --- list --- uint8 - player id uint32 - ping in nanoseconds uint32 - jitter in nanoseconds

From:	Server
To:	Client
Packet type:	Server: Keepalive packet
Enum Identifier:	SERVER_KEEPALIVE
Description	Sent periodically to keep the connection alive, in case there is no other traffic on the channel. I.e. sitting in a lobby for example.
Payload	uint8 - PACKET ID

4.4 Master server

From:	Server
To:	Master Server
Packet type:	Request listing challenge
Enum Identifier:	SERVER_REQUEST_LISTING
Description	1st step towards getting listed to the master server list
Payload	uint8 - PACKET ID string - server name uint16 - port used

From:	Master Server
To:	Server
Packet type:	Master server: listing challenge reply
Enum Identifier:	MASTER_SERVER_CHALLENGE
Description	Send a session key to the server that it needs to use if it wants to get listed on the server list.
Payload	uint8 - PACKET ID uint32 - session key to use

From:	Server
To:	Master Server
Packet type:	Server: update info
Enum Identifier:	SERVER_UPDATE_INFO
Description	Sent every 30 seconds, and every time a client joins / disconnects to update the server status on the master server list.
Payload	uint8 - PACKET ID uint32 - session key uint8 - current player count uint8 - max player count bool/uint8 - is game running?

From:	Master Server
To:	Server
Packet type:	Master server: error
Enum Identifier:	MASTER_ERROR
Description	Sent in cases when there was an error with the sent data.
Payload	uint8 - PACKET ID uint8 - error reason

From:	Client
To:	Master Server
Packet type:	Client: request info
Enum Identifier:	CLIENT_GET_SERVERS
Description	Sent to the master server in hopes to receive the list of available servers registered to the master server list.
Payload	uint8 - PACKET ID

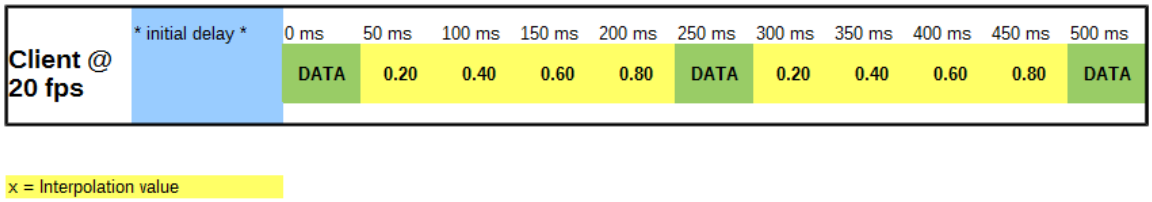
From:	Master Server
To:	Client
Packet type:	Master server: give info
Enum Identifier:	MASTER_SERVER_LIST
Description	Server information sent from the master server list.
Payload	uint8 - PACKET ID uint16 - number of servers in this packet --- list --- string - server name bool/uint8 - uses ipv6? 16 bytes - ipv6 address uint16 - server port uint8 - player count uint8 - max players bool/uint8 - game running?

5. Advanced features

5.1 Real time game

As mentioned in the implementation details section, the game internally uses a 250 millisecond refresh rate in its current implementation. This logic refresh rate is independent from the client's graphical refresh rate / frames per second performance. All the updates that are received from the server are applied gradually / interpolated on top of the last logic tick's end position, so that the motion seems smooth, even though the update rate is actually very slow. Since the positions are only interpolated between the 2 known locations, the representation of the game state is inherently always late. However, as the game isn't very fast paced in nature, we didn't see this as a problem.

Also, the current system makes it really easy to keep the clients in sync. Every client gets the same data to simulate from the server's end, and the clients do not do any kind of extrapolation that should be corrected. When a client successfully simulates a tick, it sends the new commands to the server. When all the command messages are sent the client sends a message, which tells the server that the client is ready to proceed to the next tick. When the server sees that all the clients have successfully simulated the assigned ticks, the server simulates the next one, and sends the data to the clients.



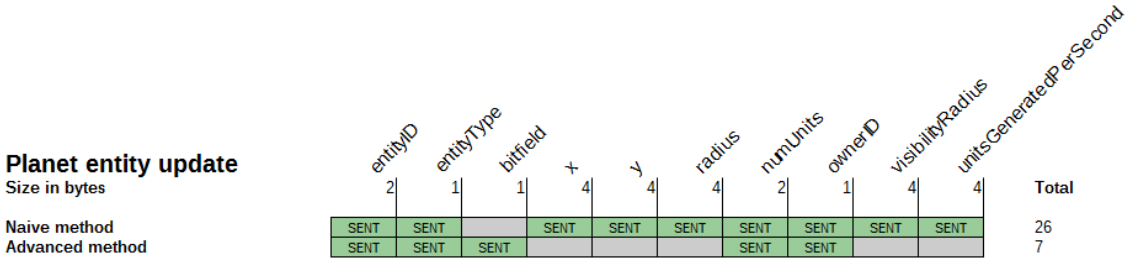
Interpolation scheme

A problem with this kind of approach is the delays caused by the networking scheme, especially when there is packet loss / resending delays involved. To avert the game from stopping at such situations, there is a small buffer to the given commands. This buffer is constructed by the clients sending their commands to be executed 2 ticks in the future, much like described in the article *1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond* [1]. This gives some leeway for the delayed scenarios, as even if the communications were cut off totally, there are buffered commands to be executed, giving more time for the correction of the situation.

5.2 Packet compression

Packet compression uses simple but effective way to reduce entity update packet sizes. The implementation uses 1 byte bit field that is sent in the entity update packet, with each bit representing each field of the entity. When entity data is changed in the server's simulation bit is set in that entity's bit field. When entities are being sent to the clients the server checks what fields are set as changed in the bit field and send only the corresponding data. This effectively removes all unnecessary entity data from update packets and thus reduces their size. Basically the tradeoff with this is just the added processing time (negligible) and added complexity of the message handling code.

Without this bit field, for example, the naive approach of updating data regarding a single planet would take total of 26 bytes of buffer space. But when this bit field is used, only the relevant information is sent, meaning that the packet data/protocol actually conforms to the data that has actually been changed. In a scenario where the planet owner and troop amounts are changed when a battle has occurred, the update size of a single planet drops from 26 bytes to total of 7 bytes, including the size of the bit field. This reduction also works for fleets where fields such as “getPlanetIDFrom” are sent only on entity creation. This kind of packet compression method is also used in the game *freeciv* that was studied earlier in assignment 2 of this course.



Packet compression scheme

5.3 Area of interest filtering

The game has an area of interest filtering that also has a good side effect of reducing data to be sent for each client. This filtering works by going through all of the entities and only sending the data regarding those that the player can currently see, based on the visibility radiuses of the entities (planets, fleets) that the player “owns”. Sent data is reduced drastically because only the information about the entities that are inside the player’s area of vision is sent. This method however requires us to store the states of the entities for each player separately, as they are sent. This is to make sure there aren’t scenarios where the client has not seen the creation of an entity, and would be receiving updates for an entity that from the client’s perspective, does not exist. This is achieved by keeping a map for each entity where an information about the last tick when the entity in question was updated for each of the players in the game. This kind of list allows us to check for scenarios when the entity data has to be updated differently for different players. If the player has not ever received an update from the entity, but it has just now reached the area of interest of the player, we know that we need to send an initialization packet

first. Also, when used in a conjunction of the information about the last time an entity was updated, this list can be used to check whether old instances a client is aware about have changed while they have been out of the client's vision range.

Technically this area of interest filtering is based on the visibility radiuses of the entities. As all the visibility areas are just simple circles, the filtering can be done with simple checks between the distances of the entities in question. No advanced algorithms are in place for this, other than a simple going through of the list of the owned entities of a player, and checking that should we update all the other entities' information based on the fact if they are close enough to the entities owned by the player in question. This "brute force" -method could become a performance concern with a high tick rate, and a much higher amount of entities in play. For the current implementation this approach was not seen as a problem, though.

This filtering also prevents "map hack" cheating where player could reveal all of the game world by investigating the network communications. The reason for this is because players only receives data about entities that they can see, so there is nothing to be revealed. This kind of cheat prevention would be impossible to achieve for all clients if the game was not using a dedicated server, because the hosting player would have to know the full game state.

5.4 Fog of war

Implementation of fog of war is mostly for client side effects like shading non-visible areas darker and not rendering fleets when they are not within the players vision (not receiving updates). Players can still see all of the initial planet locations and their sizes as they are sent from the server upon the game start. The area of interest filtering system would allow us to implement true fog of war also, where the player would not even know about these locations of the existing planets before scouting out for them, but this would require us to implement a more advanced input scheme, where the fleets could be individually controlled and moved freely around the map.

Currently there are some bit more complicated rules for the planet visibility; if a planet is neutral, its size information is displayed regardless of the fog. But, if a planet is controlled by another player, its information is visually represented to be unknown by marking the unit amount with "?". This still means that the neutral planets can change ownership so that a player does not

even know about it. This distinction about the unit amounts displayed is just made so that the game is more readable and less confusing for the player. For fleets similar rules apply. Players can only see fleets that are within their vision. If fleet enters and then later exits players vision the player loses track of the fleet and it fades away (the visual fleet representation fades away when no update packets are received for a period of 2 seconds).

This kind of implementation means that players can't see where other players are sending fleets or what planets they have conquered. However this does not include the homeworld planet that the players start with. This decision was made so that players don't have to search for other players thus increasing action.

The fog of war is tightly coupled with the area of interest filtering because the visual representation state of the entities is based on the update data received from the server. This update data is coupled with the area interest filtering, which on the other hand is based on the "visibility radiuses" of each of the entities that can be seen on the client's side as the lighter shaded areas of the world.

It is worthwhile to mention that even though the updates for the fleets are stopped and the fog-of-war system stops representing the fleets visually at this point – technically however as the fleets move at constant speed, and their movement cannot be currently altered mid-flight, the fog system just masks information away from the player. Since the fleets move at a constant speed and they don't change their targets, it might make sense for someone to generate a cheat that would keep the fleets visible for the whole range of their flight after they are initially spotted by the player. However, we thought that this would give the players minimal advantage, and solving this problem would cause more unneeded technical complexity to the game. Also, if an option to change the course of the fleets in mid-flight was to be added, this problem would not exist / play such a big role anymore.

6. Work schedule

Milestones

1. Groups have to be formed — 24.11.2013 23:59
   - Create initial game design prototype ✓
   - Prototype the game idea in action ✓
   - Create initial game design document ✓

2. Initial game idea and design document — 29.12.2014 23:59
   - Finalize game design ✓
   - Set up version controlling system (git) ✓
   - Write project code ✓
     - Most of the client side functionality ✓
     - Simple server implementation ✓

3. Mid return (access to codes) — 30.01.2015 12:00
   - Finalize documentation ✓
   - Master server ✓
   - Finalize client and server ✓
   - Testing ✓

4. Document return — 16.02.2015 12:00
   - Prepare for presentation

5. Presentation day — 17.02.2015 10:00 →
   - Final bug fixes / code cleanup
   - Implementation of left out advanced features

6. Final code return — 27.02.2015 23:59