Project Darkstar:

A Case Study in Developing Games with Project Darkstar

Owen Kellett
Sun Microsystems
Project Darkstar Staff Engineer

September 15, 2008

Overview

- What is Project Darkstar?
  - Motivations for its design
  - Problems that it solves
- Example application: Project Snowman
  - Overview of game design
  - Implementation with Project Darkstar

What is Project Darkstar?

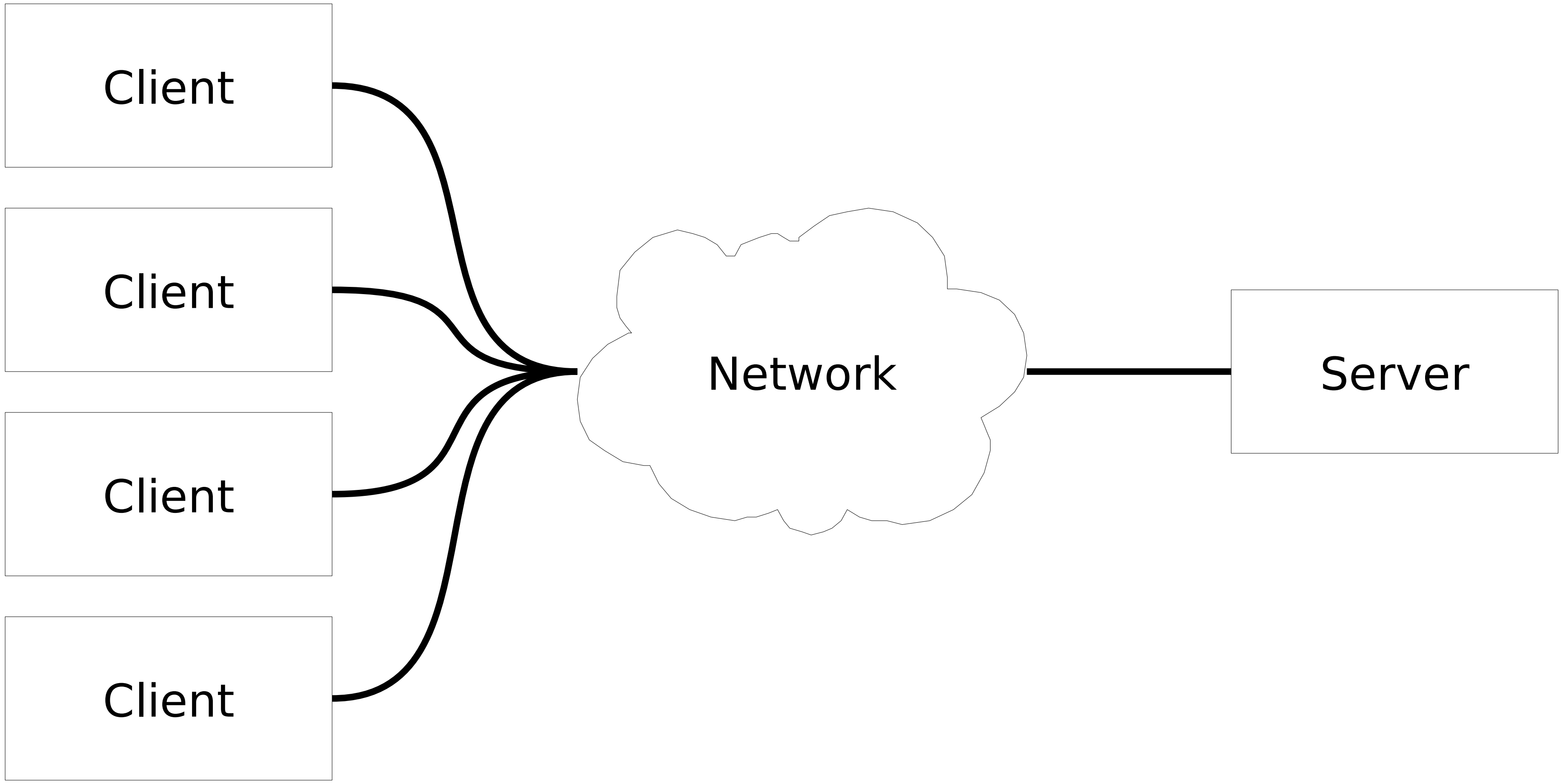
(Image From *Star Trek: Generations*)

Project Darkstar Overview

- Project Darkstar is a software platform that simplifies the development of multiplayer online games
- Written entirely in Java
- Automatically handles many infrastructure requirements on the server side of such games
  - Communications
  - Thread management
  - Contention management
  - Persistence
  - Automatic Scaling

Project Darkstar

- Goal: Write a networked multiplayer game
- Problem: Where do I start?

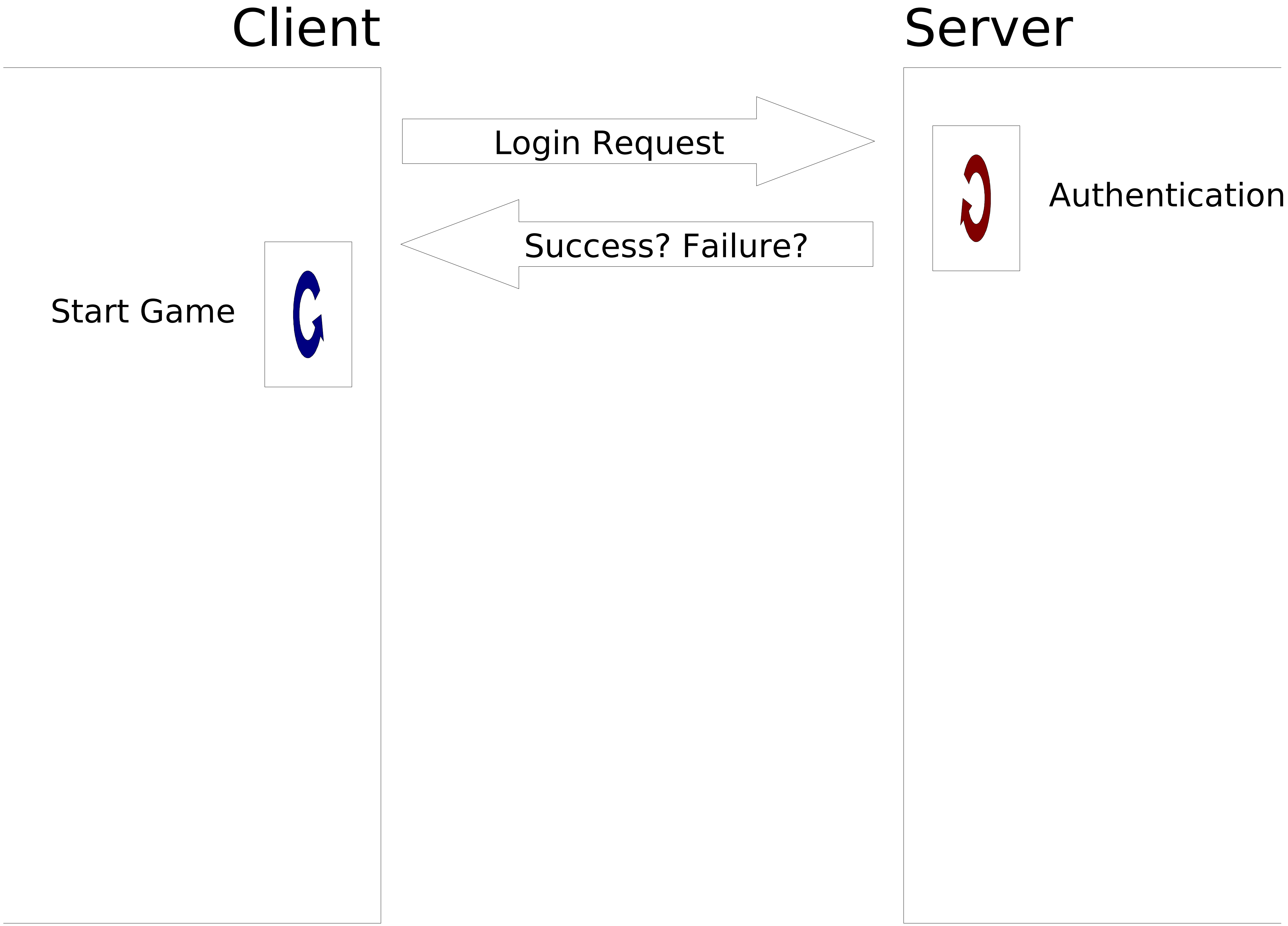


(Useful Diagram)

Project Darkstar

- Observation: Most games share the same coarse grained tasks between the client and server
- What are they?

Client
?
Server



Client

Server

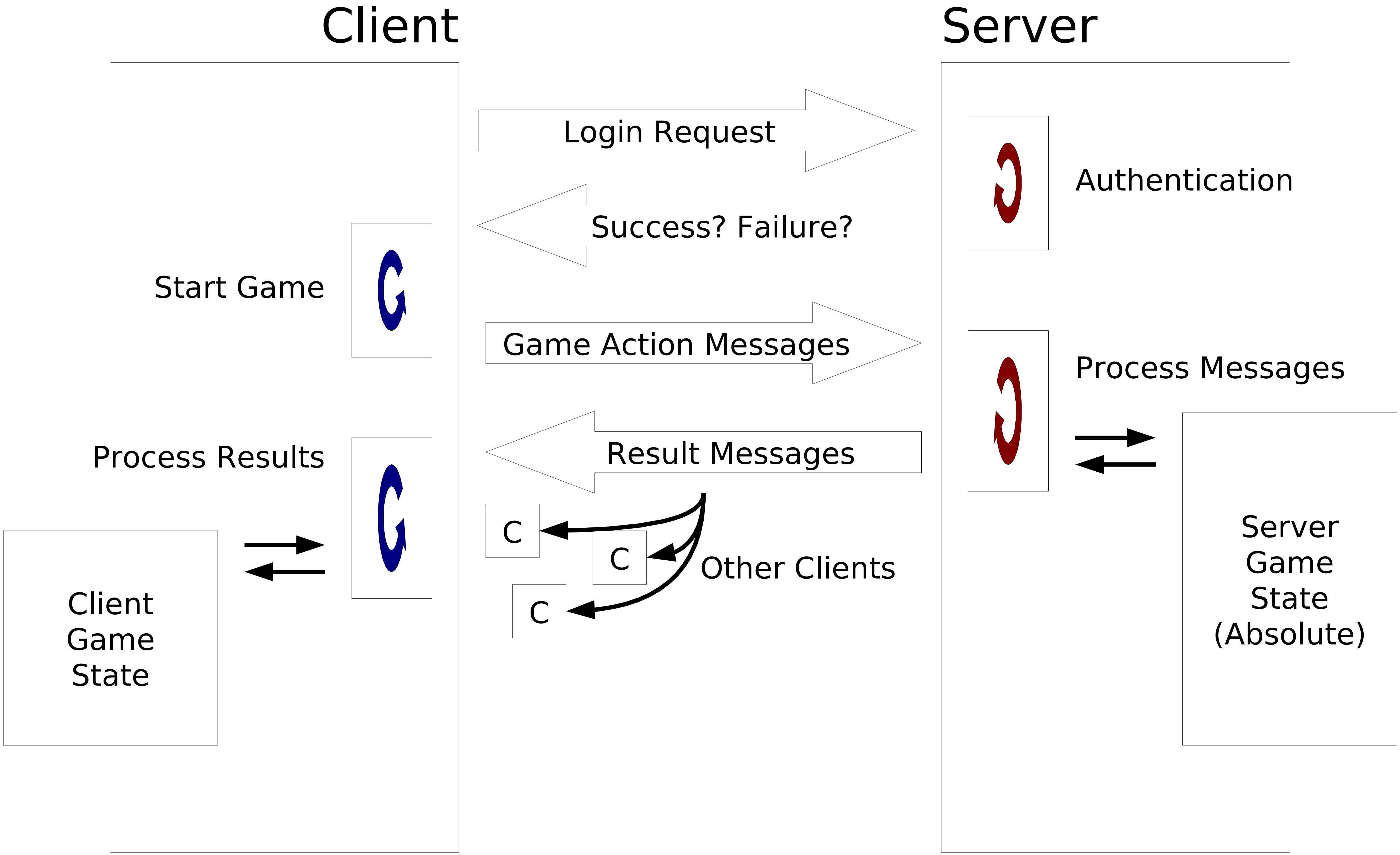
Login Request

Authentication

Success? Failure?

Start Game

Project Darkstar



Client

Server

Login Request

Authentication

Success? Failure?

Start Game

Game Action Messages

Process Messages

Result Messages

Process Results

C

C

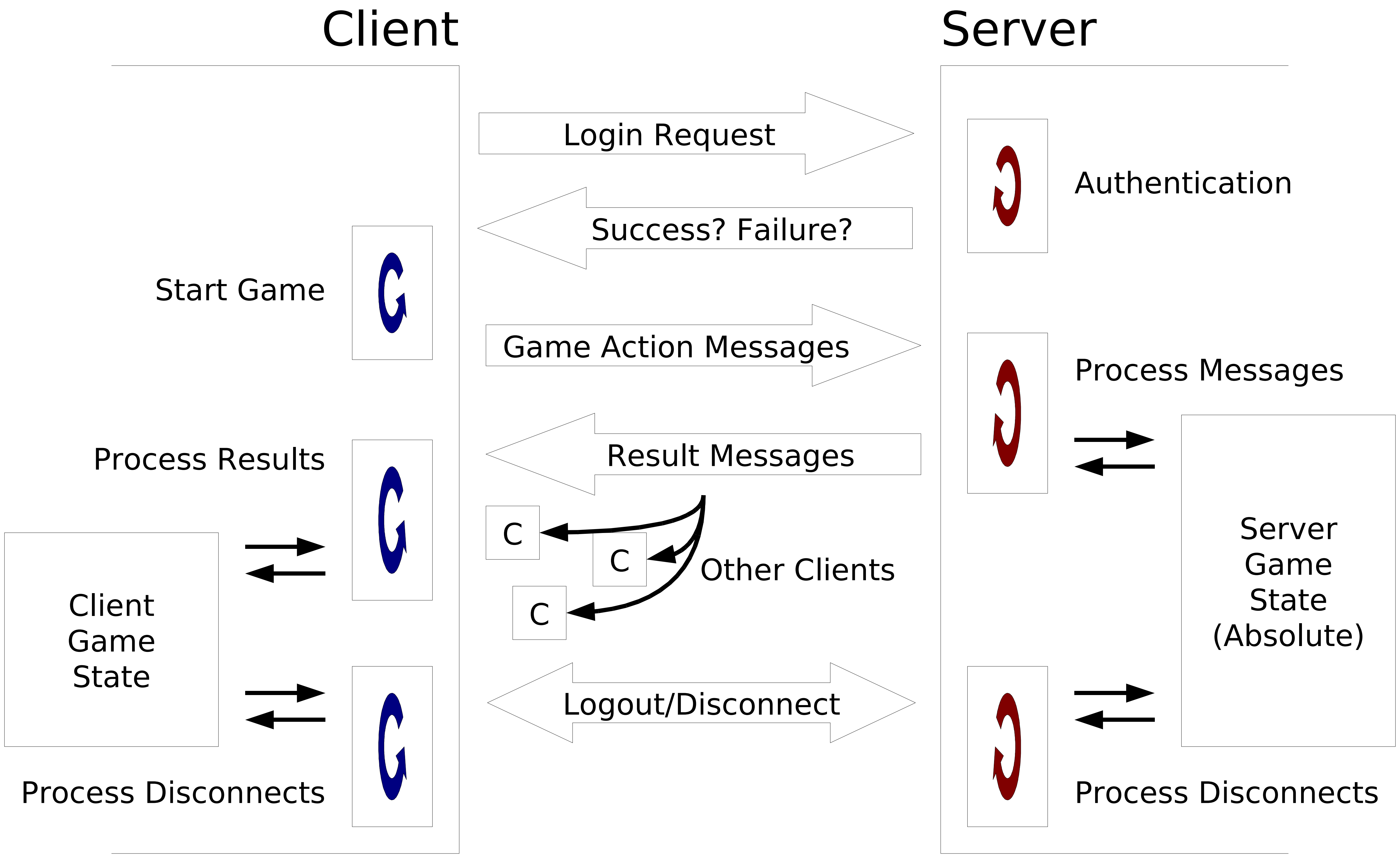
C

Other Clients

Client Game State

Server Game State (Absolute)

Project Darkstar



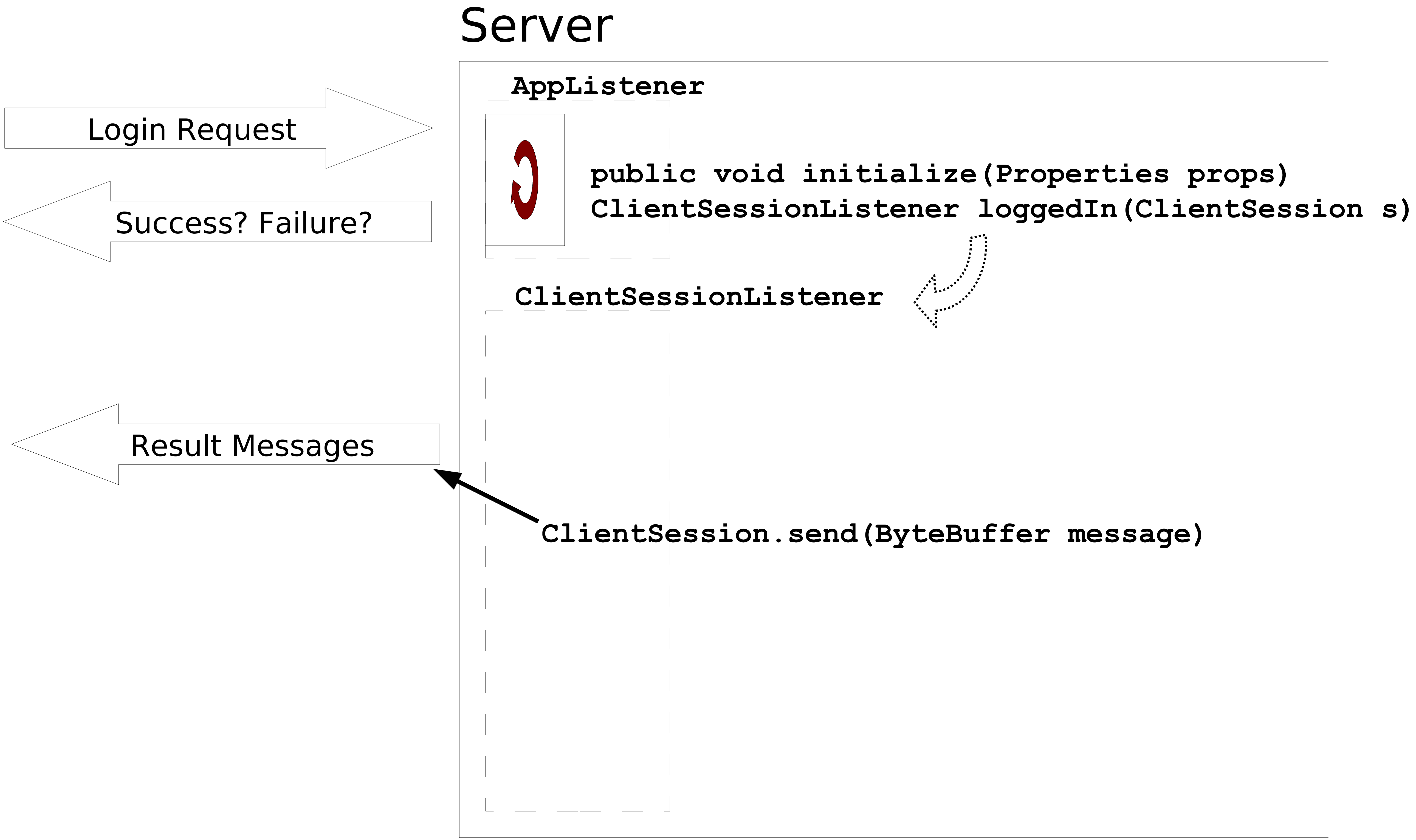
Problem 1: Communications

- Just setting up a system to handle logins and messages is hard work
- Network Programming?
- Sockets? RMI?
- I don't want to do that, I want to write a *Game!*

Get Communications back online!
I can't sir. The rmiregistry is down.

Problem 1: Communications

- Project Darkstar abstracts away *ALL* Network Programming mechanics
- Provides intuitive API to handle all of the coarse grained behavior of the communication between client and server shown previously
- How easy is it? Three interfaces on the server side:
  - **`AppListener`** (initialization and logins)
  - **`ClientSessionListener`** (receiving messages)
  - **`ClientSession`** (sending messages)



Server

Login Request

Success? Failure?

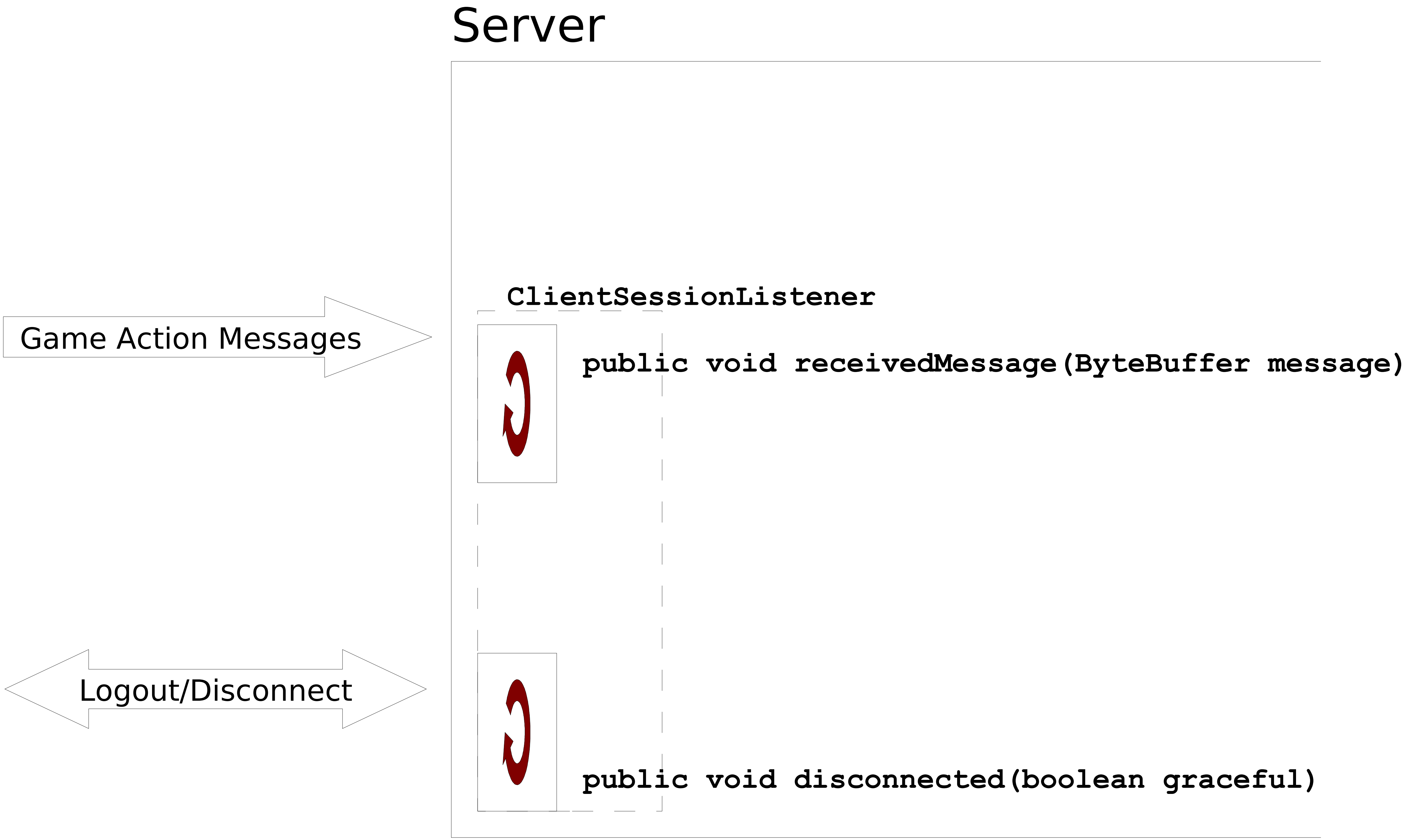
Result Messages

AppListener

```
public void initialize(Properties props)
ClientSessionListener loggedIn(ClientSession s)
```

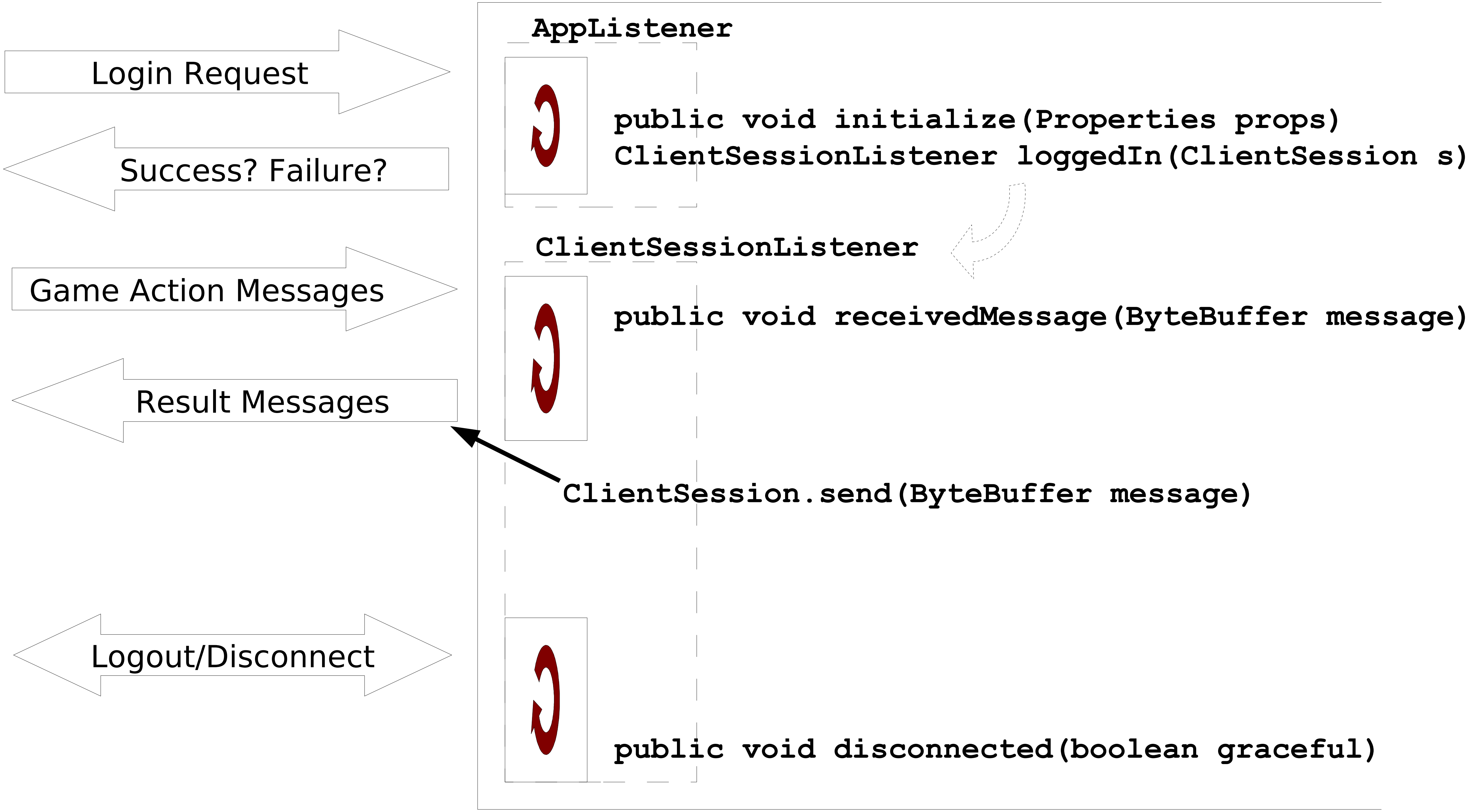
ClientSessionListener

```
ClientSession.send(ByteBuffer message)
```



```
public void receivedMessage(ByteBuffer message)
public void disconnected(boolean graceful)
```

Server



Login Request

Success? Failure?

Game Action Messages

Result Messages

Logout/Disconnect

AppListener

```
public void initialize(Properties props)
ClientSessionListener loggedIn(ClientSession s)
```

ClientSessionListener

```
public void receivedMessage(ByteBuffer message)
```

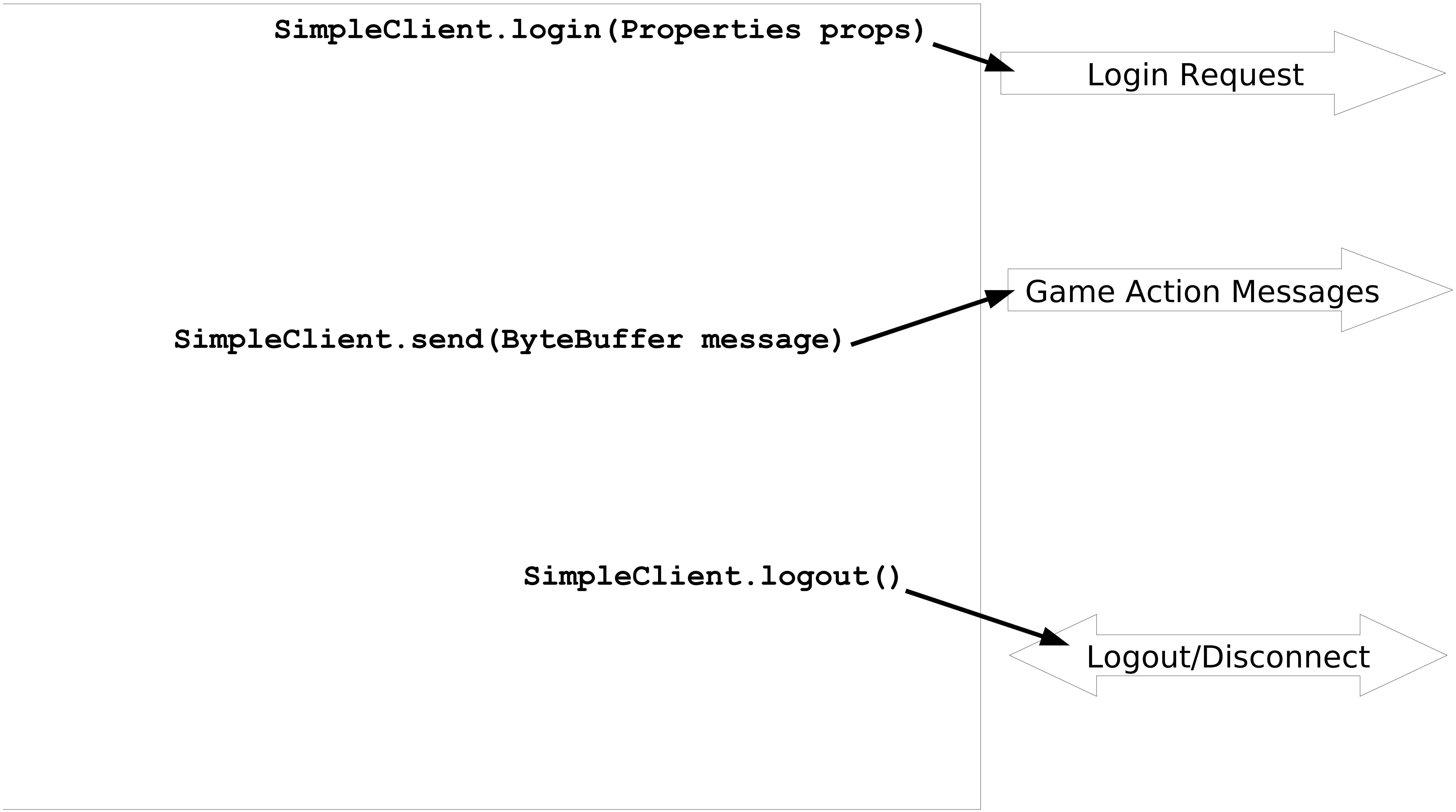
```
ClientSession.send(ByteBuffer message)
```

```
public void disconnected(boolean graceful)
```

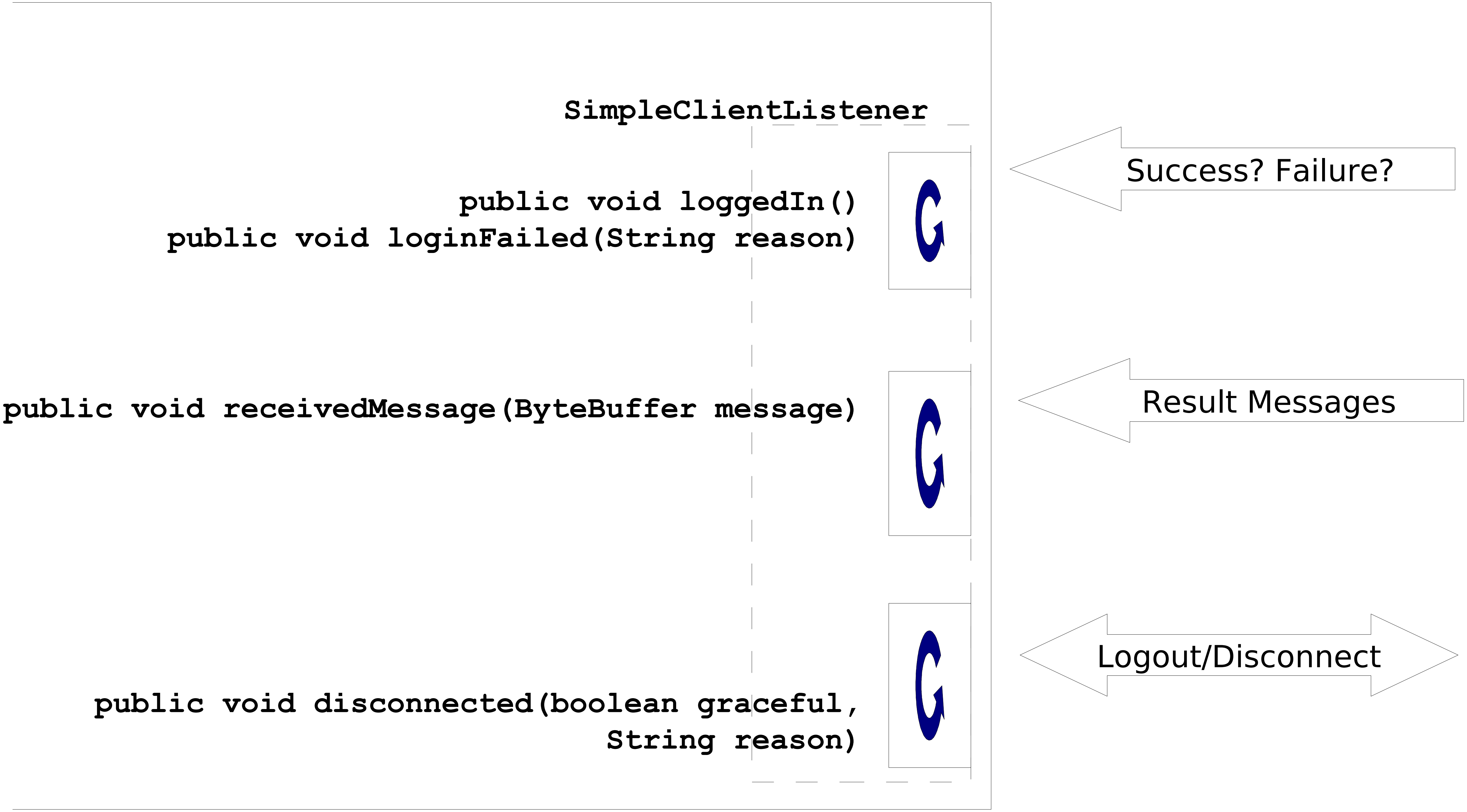
Problem 1: Communications

- What about the client?
- Language agnostic
  - A client API can be easily implemented in any language by conforming to the wire protocol
- Our Java implementation? One class and one interface:
  - **`SimpleClient`** (logins/logouts and sending messages)
  - **`SimpleClientListener`** (receiving messages and login success/failure notifications)

Client


SimpleClient.login(Properties props)
Login Request
Game Action Messages
SimpleClient.send(ByteBuffer message)
SimpleClient.logout()
Logout/Disconnect

Client



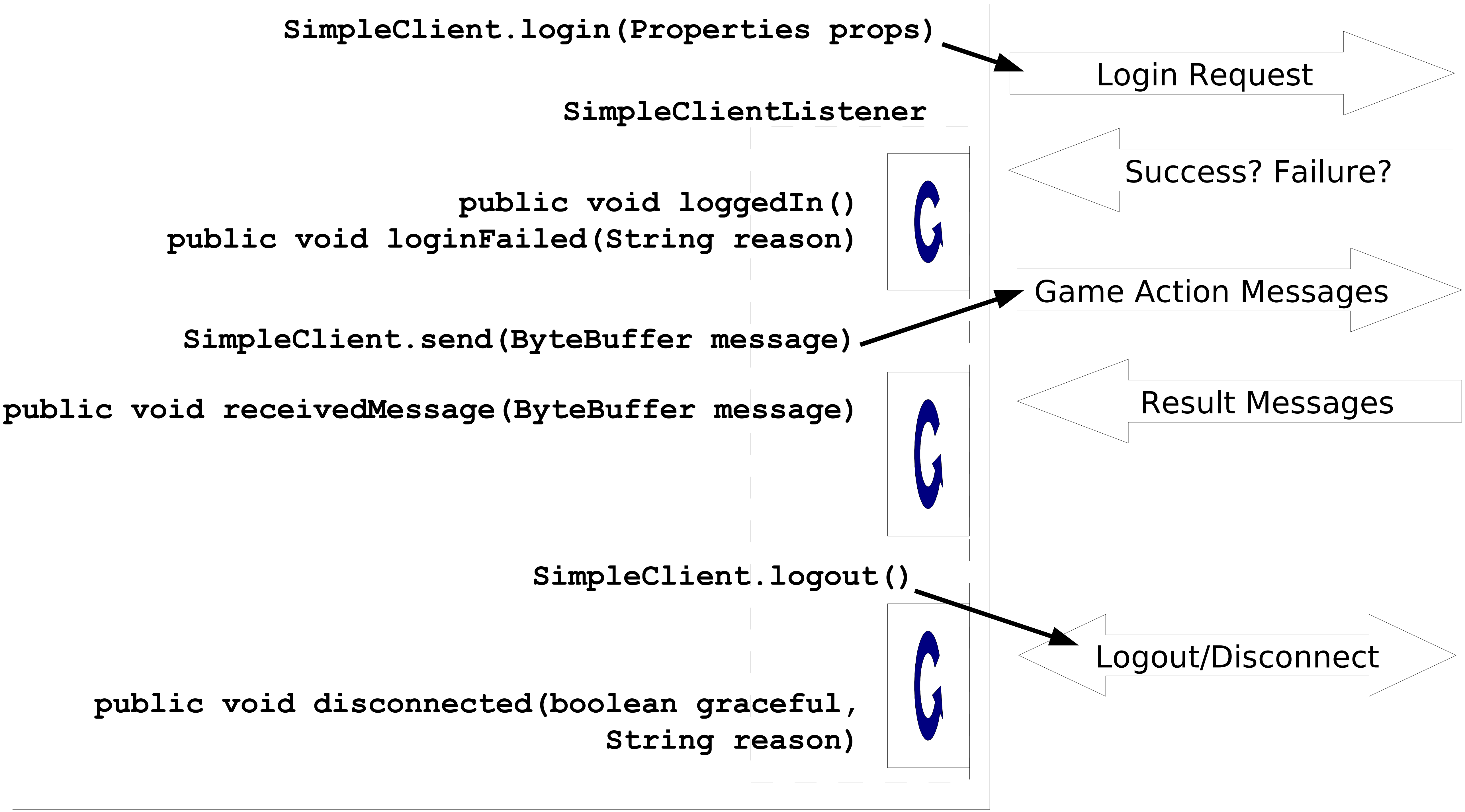
```
SimpleClientListener

public void loggedIn()
public void loginFailed(String reason)

public void receivedMessage(ByteBuffer message)

public void disconnected(boolean graceful,
                         String reason)
```

Success? Failure?

Result Messages

Logout/Disconnect

Client



SimpleClient.login(Properties props)

SimpleClientListener

public void loggedIn()

public void loginFailed(String reason)

SimpleClient.send(ByteBuffer message)

public void receivedMessage(ByteBuffer message)

SimpleClient.logout()

public void disconnected(boolean graceful,
String reason)

Problem 2: Multi-Client Communications

- What about communicating messages to multiple clients?
- Project Darkstar provides a mechanism that will batch send messages to groups of clients
- Referred to as Channels

(Image From Star Trek: TNG *Parallels*)

Problem 2: Multi-Client Communications

- Example scenarios:
  - Multiple, isolated games
  - Separate teams with isolated communications or chat messages
  - You enter a dungeon and now need to receive messages about what's going on
  - etc..

Problem 2: Multi-Client Communications

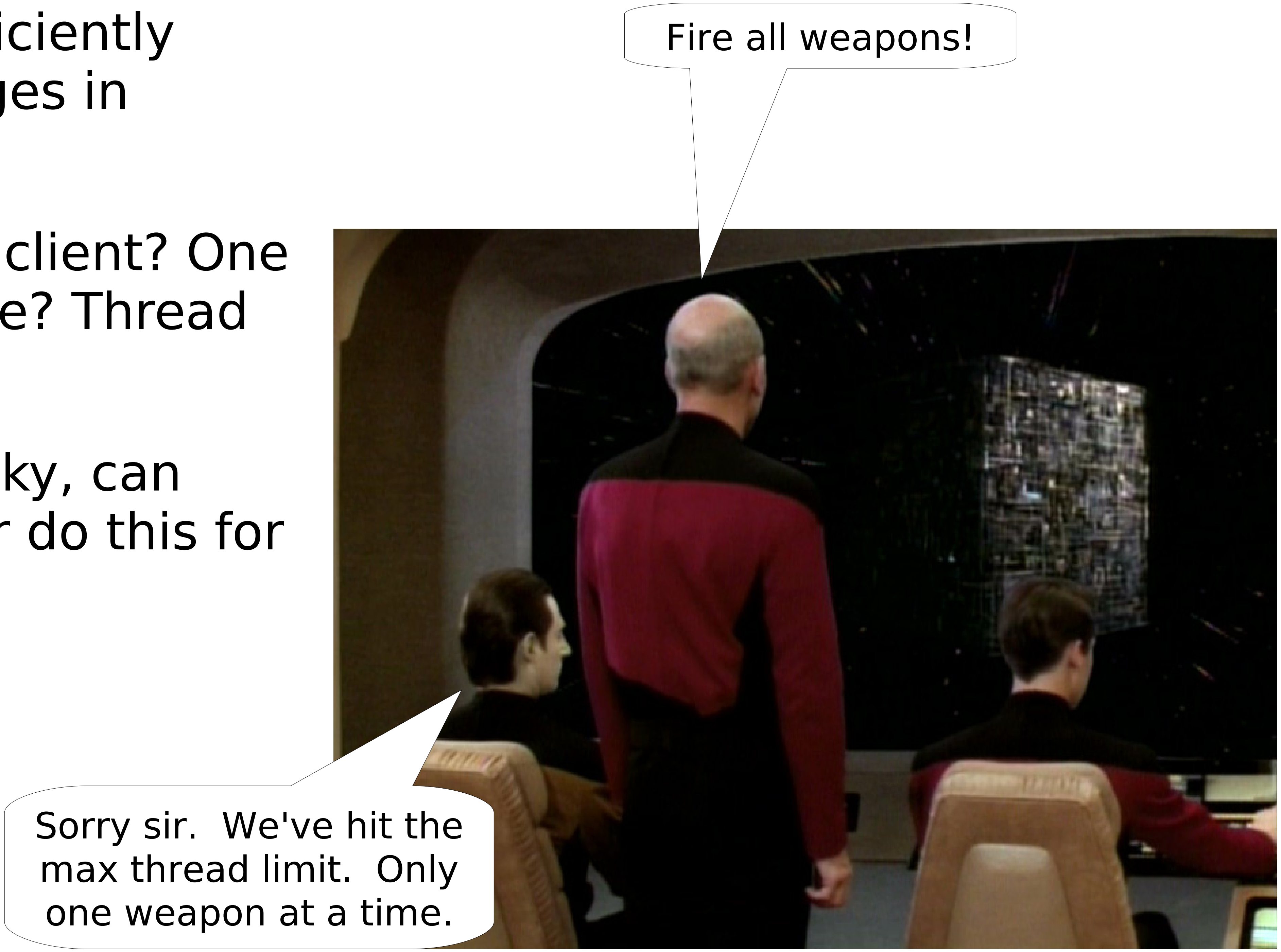
- **`AppContext.getChannelManager();`**
- **`ChannelManager`** (acquired directly from the PDS stack via the static AppContext class)
  - **`createChannel(..);`**
  - **`getChannel(..);`**
- Provides mechanisms for creating and retrieving Channels

Problem 2: Multi-Client Communications

- Server side: Two interfaces.
  - **`Channel`** (object acquired from Project Darkstar stack and used to add/remove clients and send messages to all clients on Channel)
  - **`ChannelListener`** (processes incoming messages on a channel)
- Client side: Two interfaces.
  - **`ClientChannel`** (used to send messages to all clients on the channel)
  - **`ClientChannelListener`** (processes incoming messages on a channel)

Problem 3: Thread Management

- How can we efficiently process messages in parallel?
- One thread per client? One thread per game? Thread pools?
- This sounds tricky, can Project Darkstar do this for me? Yes!


Fire all weapons!
Sorry sir. We've hit the max thread limit. Only one weapon at a time.

(Image From Star Trek: TNG *The Best of Both Worlds*)

Problem 3: Thread Management

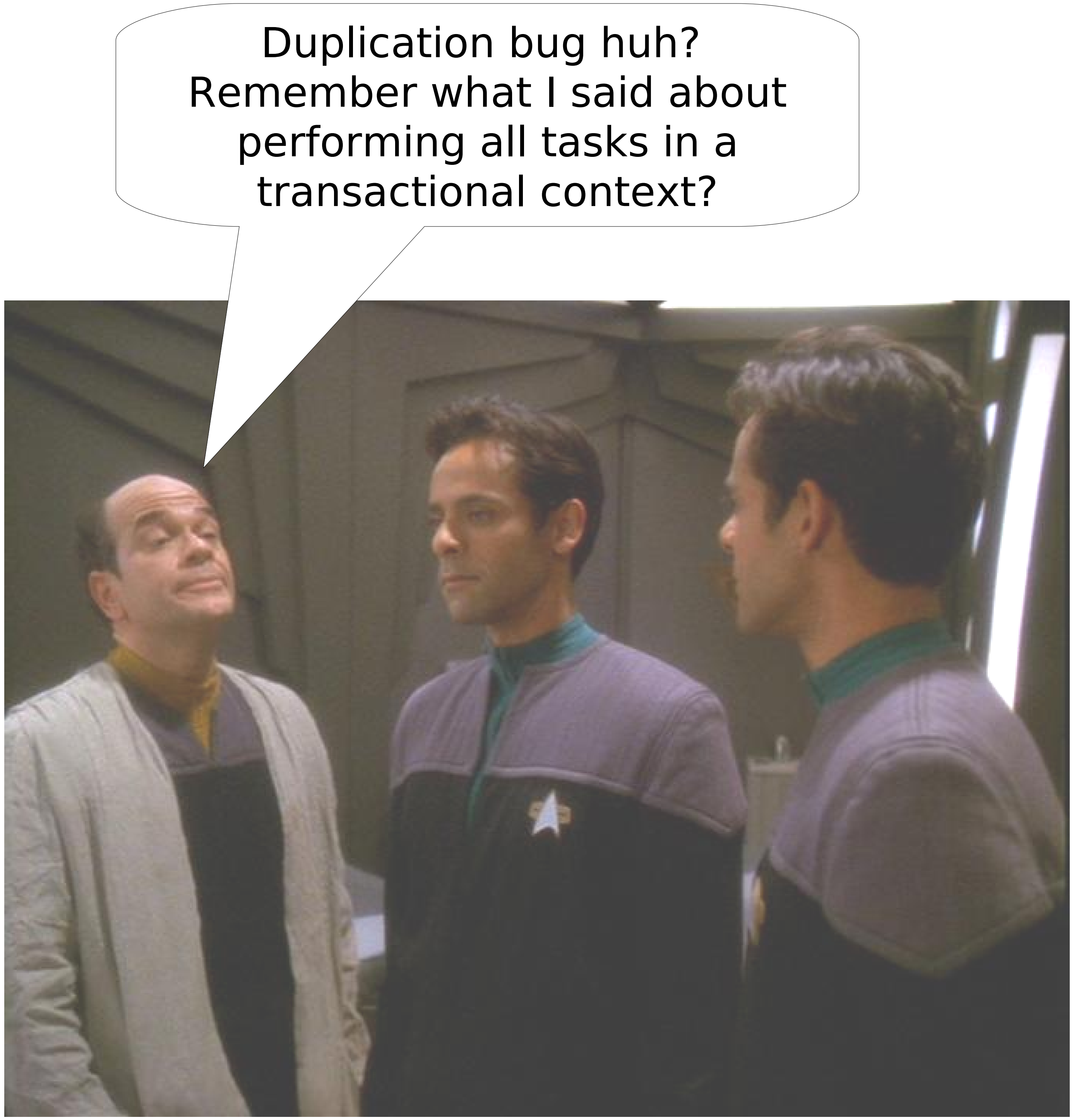
- Project Darkstar is a multi-threaded environment under the hood
- Implementing message processing game logic code appears single threaded to the developer
- Each incoming message is run in a separate task
  - **`AppListener.loggedIn(..)`**
  - **`ClientSessionListener.receivedMessage(..)`**
  - **`ClientSessionListener.disconnected(..)`**
  - **`ChannelListener.receivedMessage(..)`**
- Tasks are queued up and run by a configurable pool of threads

Problem 3: Thread Management

- **`AppContext.getTaskManager();`**
- **`TaskManager`** (acquired directly from the Project Darkstar stack via the static AppContext class)
  - **`schedulePeriodicTask(..);`**
  - **`scheduleTask(..);`**
- Provides mechanisms for scheduling your own tasks

Problem 4: Data Consistency

- With multiple threads of work operating on the same data, we need to manage potential data consistency errors.
- Project Darkstar automatically runs every task in an ACID transaction
- No explicit synchronization code required!

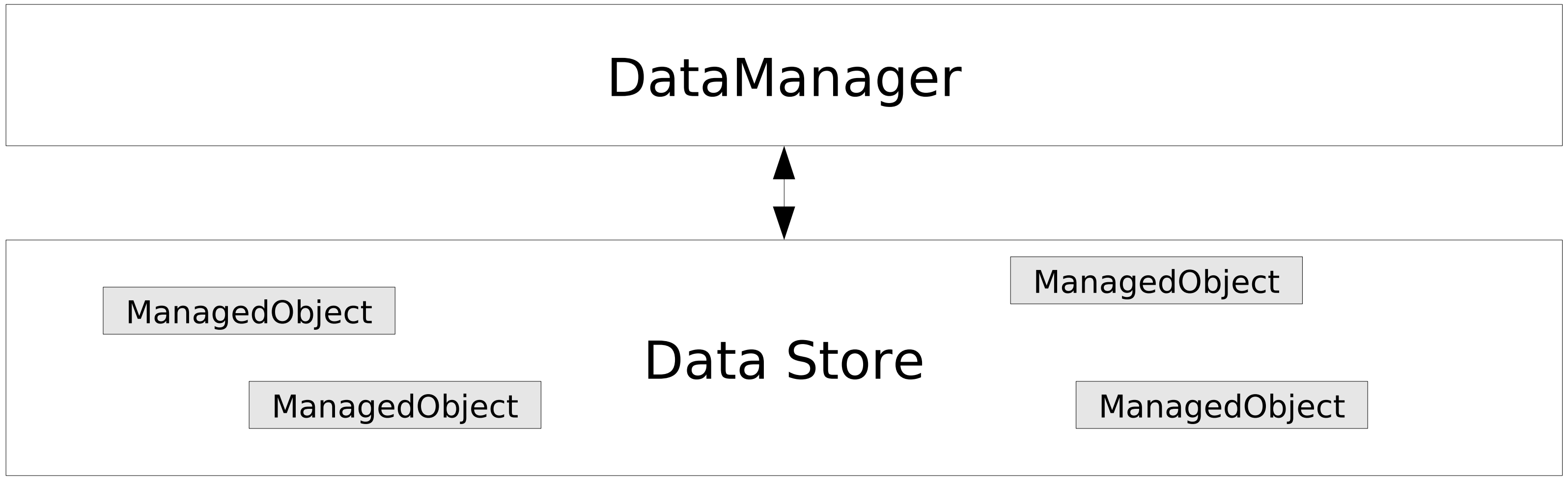

Duplication bug huh? Remember what I said about performing all tasks in a transactional context?

(Image From Star Trek: DS9 *Doctor Bashir, I Presume*)

Problem 4: Data Consistency

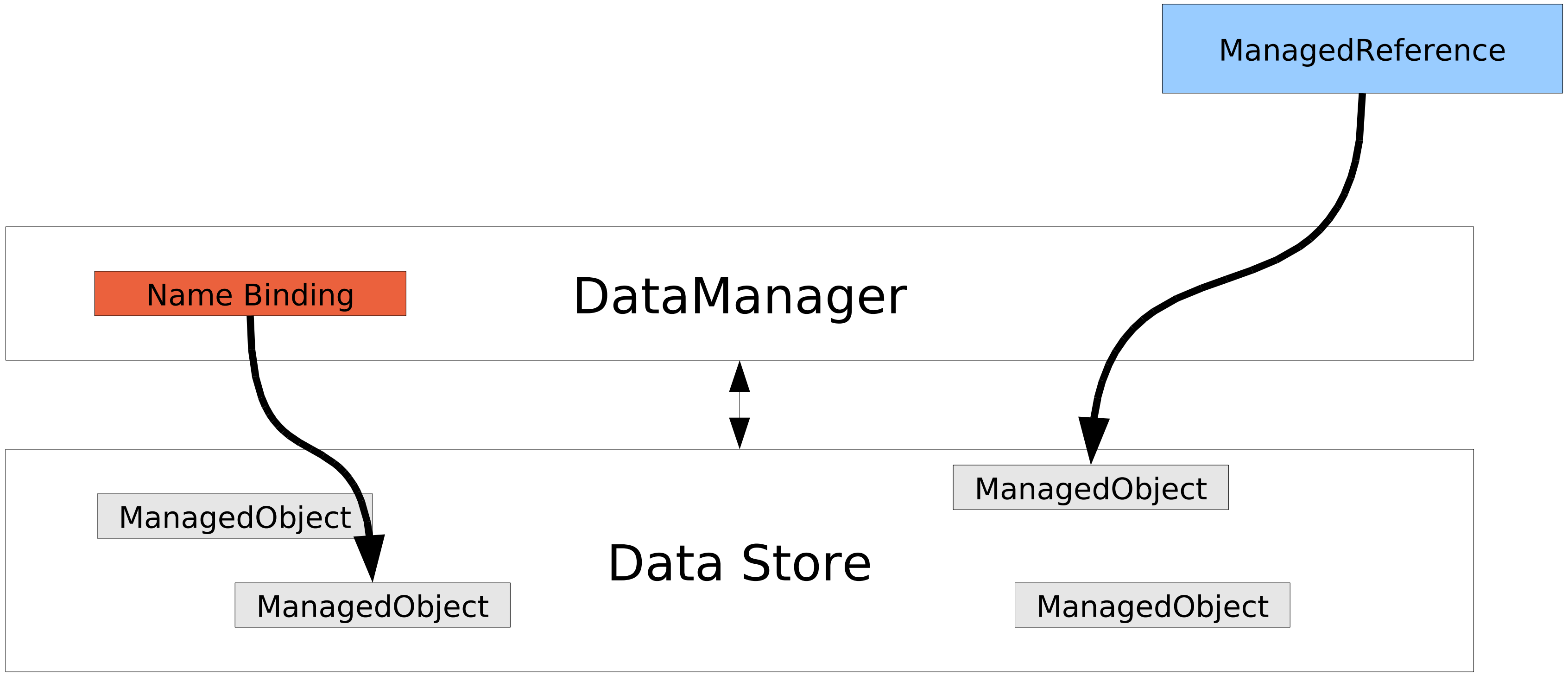
- **`AppContext.getDataManager();`**
- **`DataManager`** (acquired directly from the Project Darkstar stack via the static AppContext class)
- **`ManagedObject, Serializable`** – any shared object must implement these marker interfaces
- **`ManagedObjects`** are managed by the **`DataManager`**

Problem 4: Data Consistency



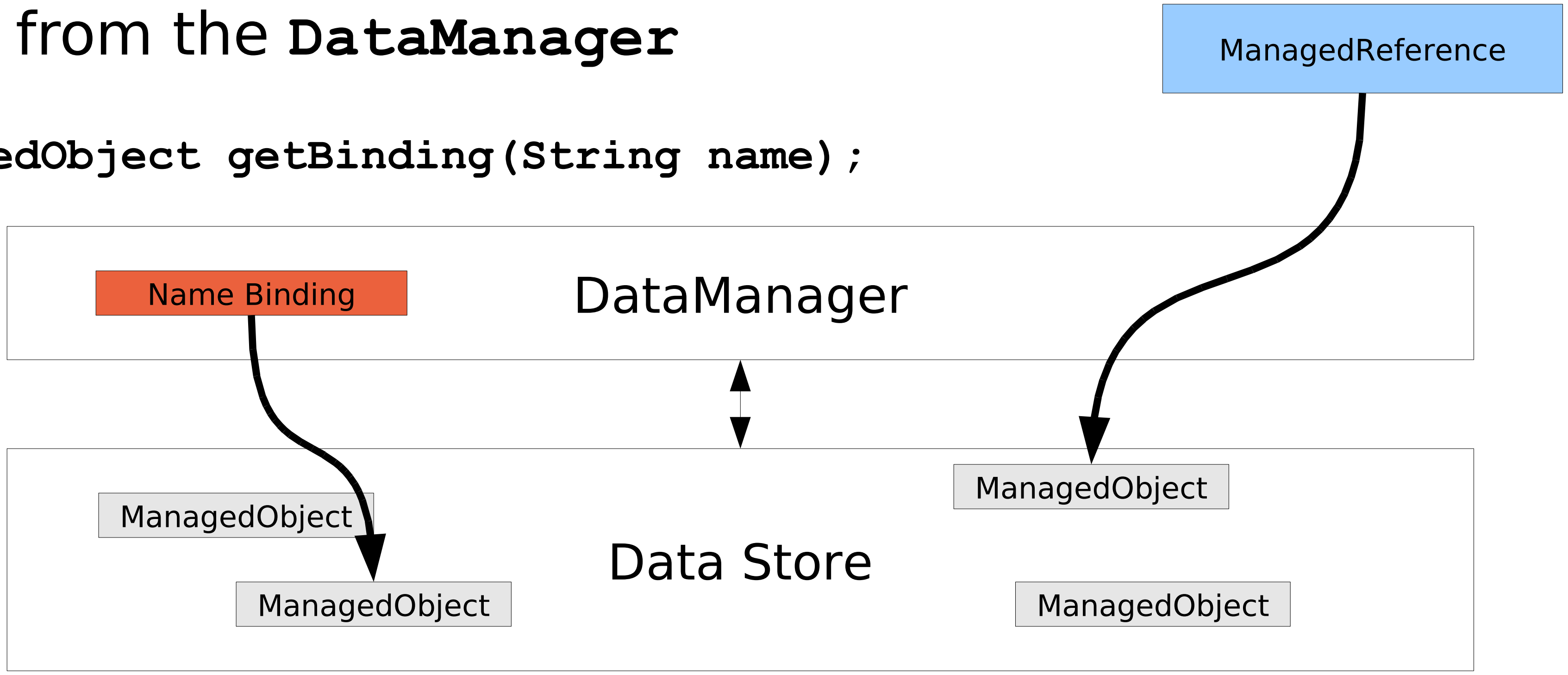
Problem 4: Data Consistency

- Two ways to save an object into the Data Store (both DataManager methods):
  - `<T> ManagedReference<T> createReference(T object);`
  - `void setBinding(String name, Object object);`



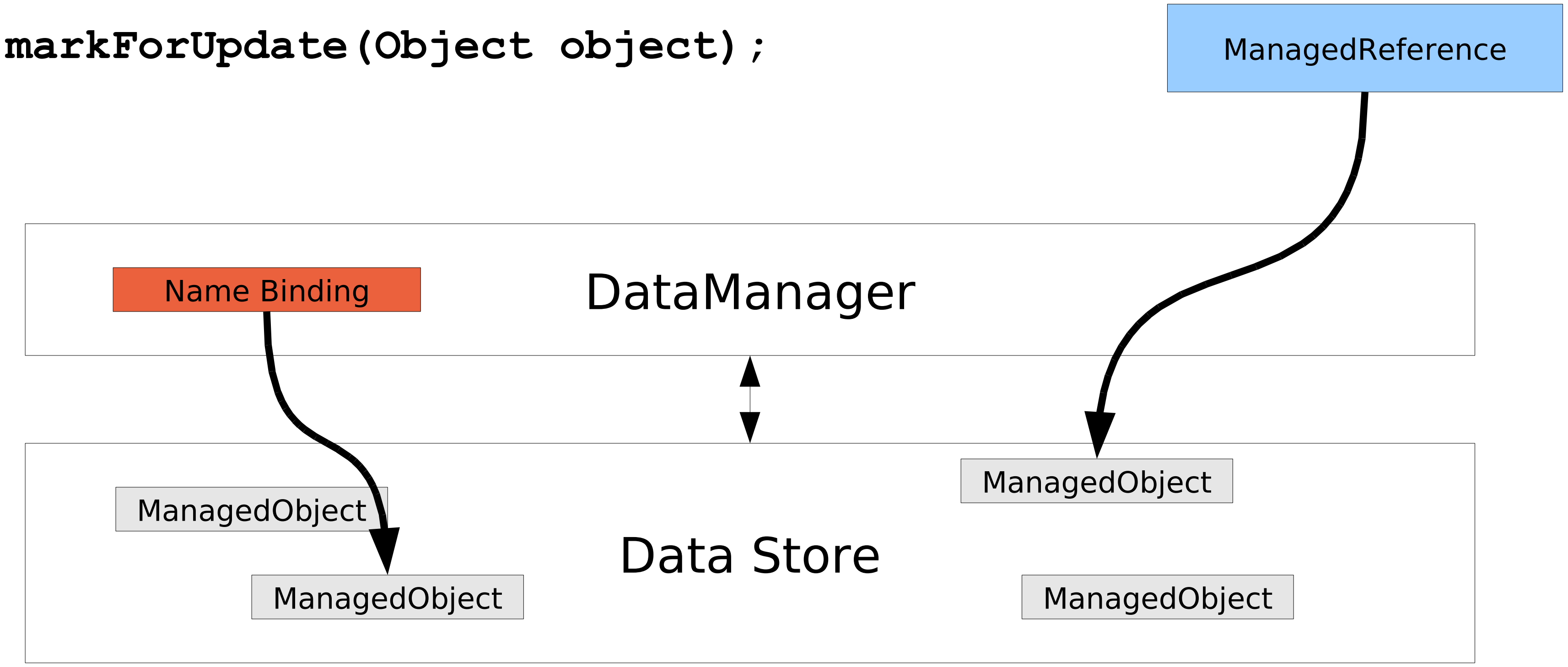
Problem 4: Data Consistency

- Three ways to retrieve an object from the Data Store:
  - From a **`ManagedReference`**
    - **`T get();`**
    - **`T getForUpdate();`**
  - Directly from the **`DataManager`**
    - **`ManagedObject getBinding(String name);`**



Problem 4: Data Consistency

- Two ways to notify the DataManager that an object is to be modified:
  - From a **`ManagedReference`**
    - **`T getForUpdate();`**
  - Directly from the **`DataManager`**
    - **`void markForUpdate(Object object);`**



Problem 5: Persistence

- When developing large virtual worlds, it is desirable to protect against server crashes and other unrecoverable problems
- Project Darkstar's default Data Store is implemented as a Berkeley DB database
- All ManagedObjects are automatically persisted to disk after every transaction!

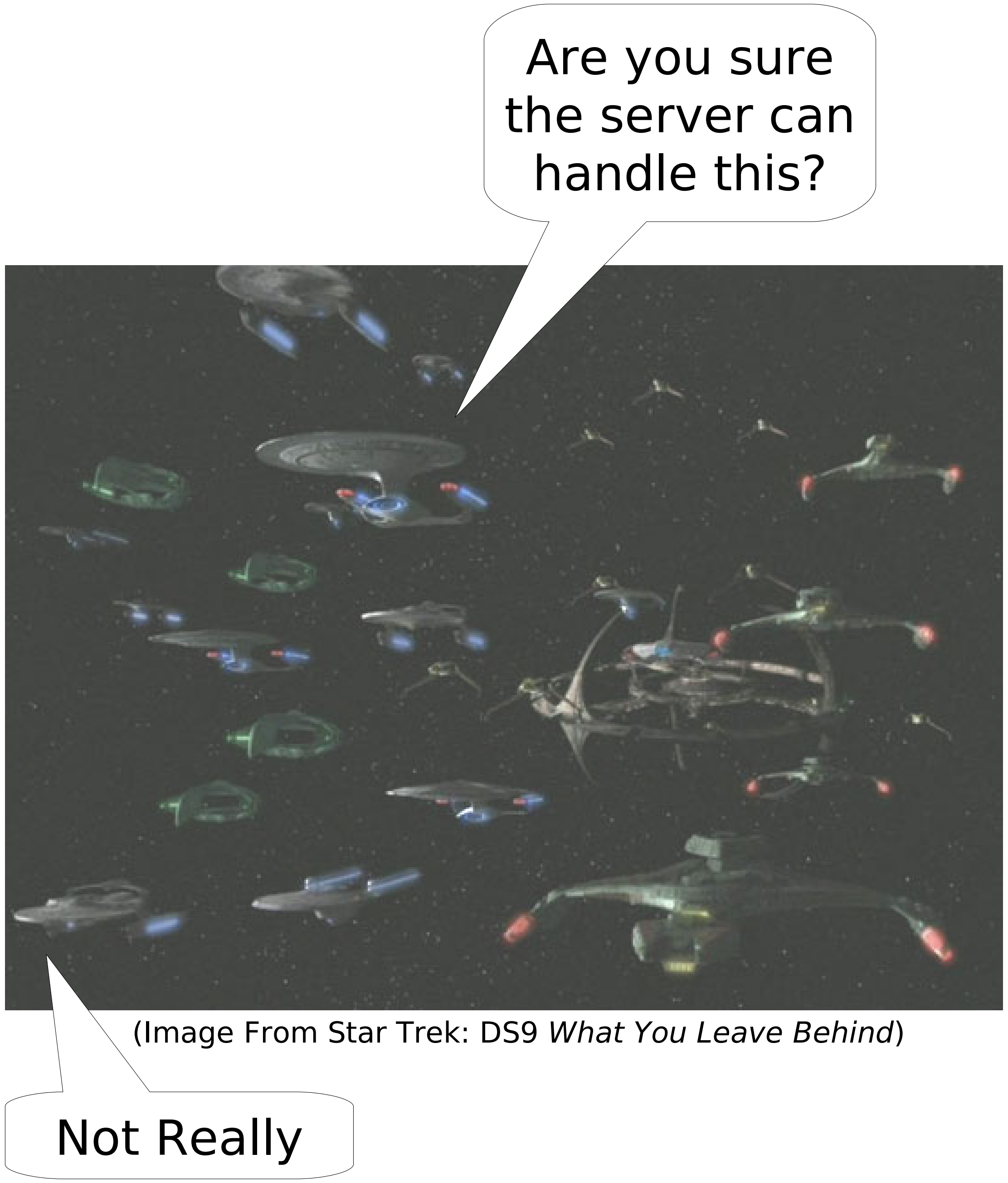
I really wish we only had to play this poker game once.

(Image From Star Trek: TNG *Cause and Effect*)

Sorry no data persistence. Every time the server crashes we have to start over.

Problem 6: Scalability

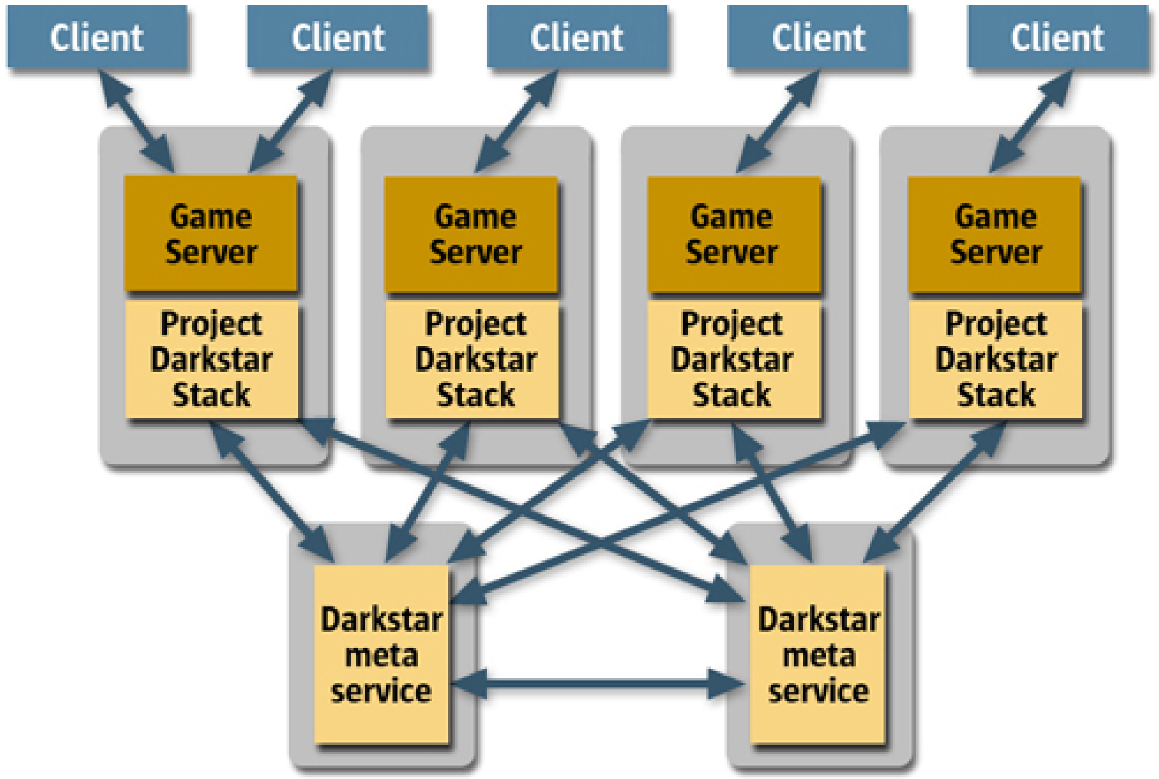
- Large virtual worlds means a lot of connected players, a lot of game state information, and a lot of server side processing.
- Current industry solution: zones and shards
- Project Darkstar supports multi-node server deployments using the same game code



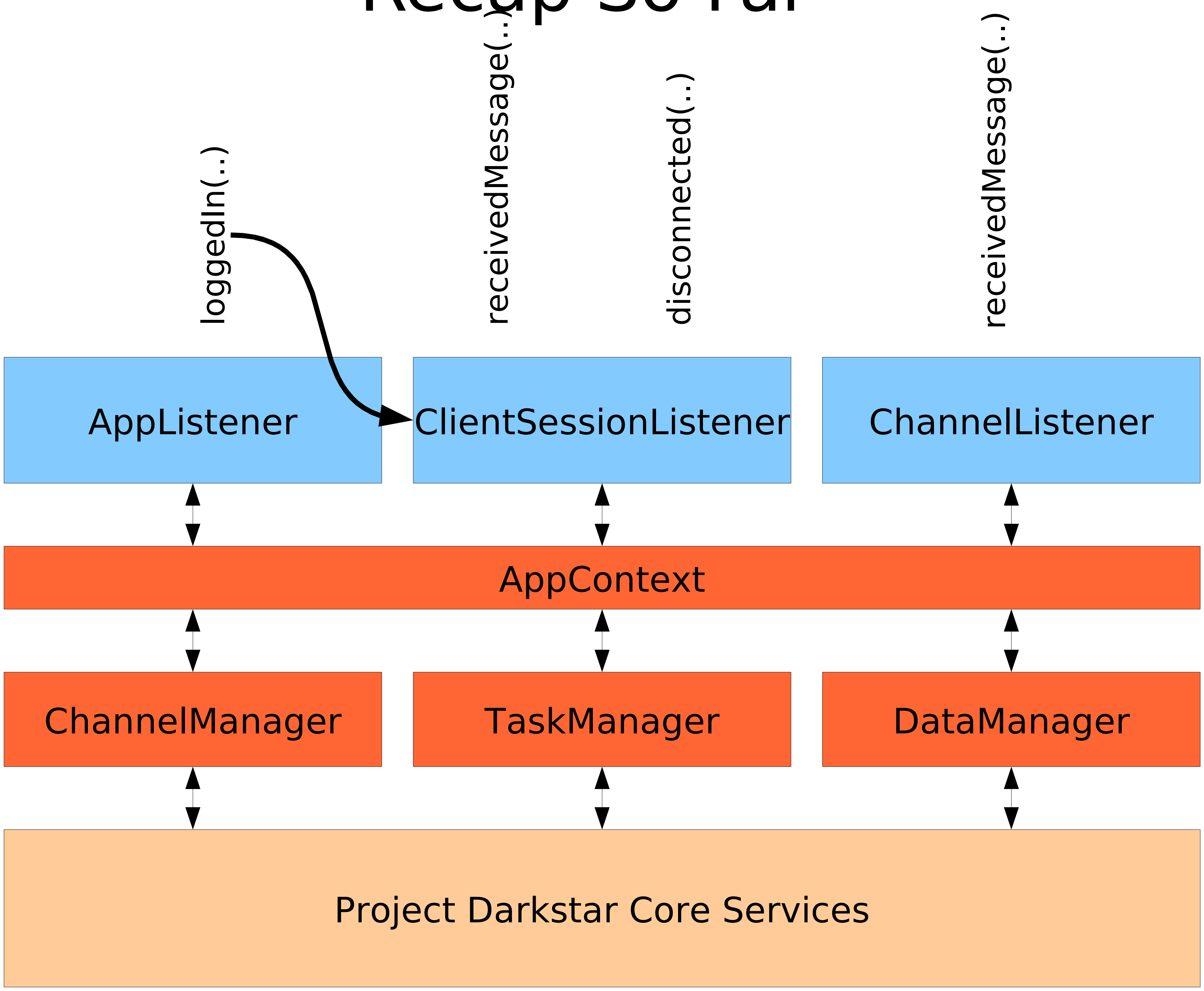
(Image From Star Trek: DS9 *What You Leave Behind*)

Problem 6: Scalability

- Difficult problems in this space
- Distributed data storage
- Automatic load balancing
- Intelligent load distribution
- Fault-tolerance and failover
- Goal: near-linear scaling
- Current Reality: fully functional multi-node system but expected performance scaling not there yet

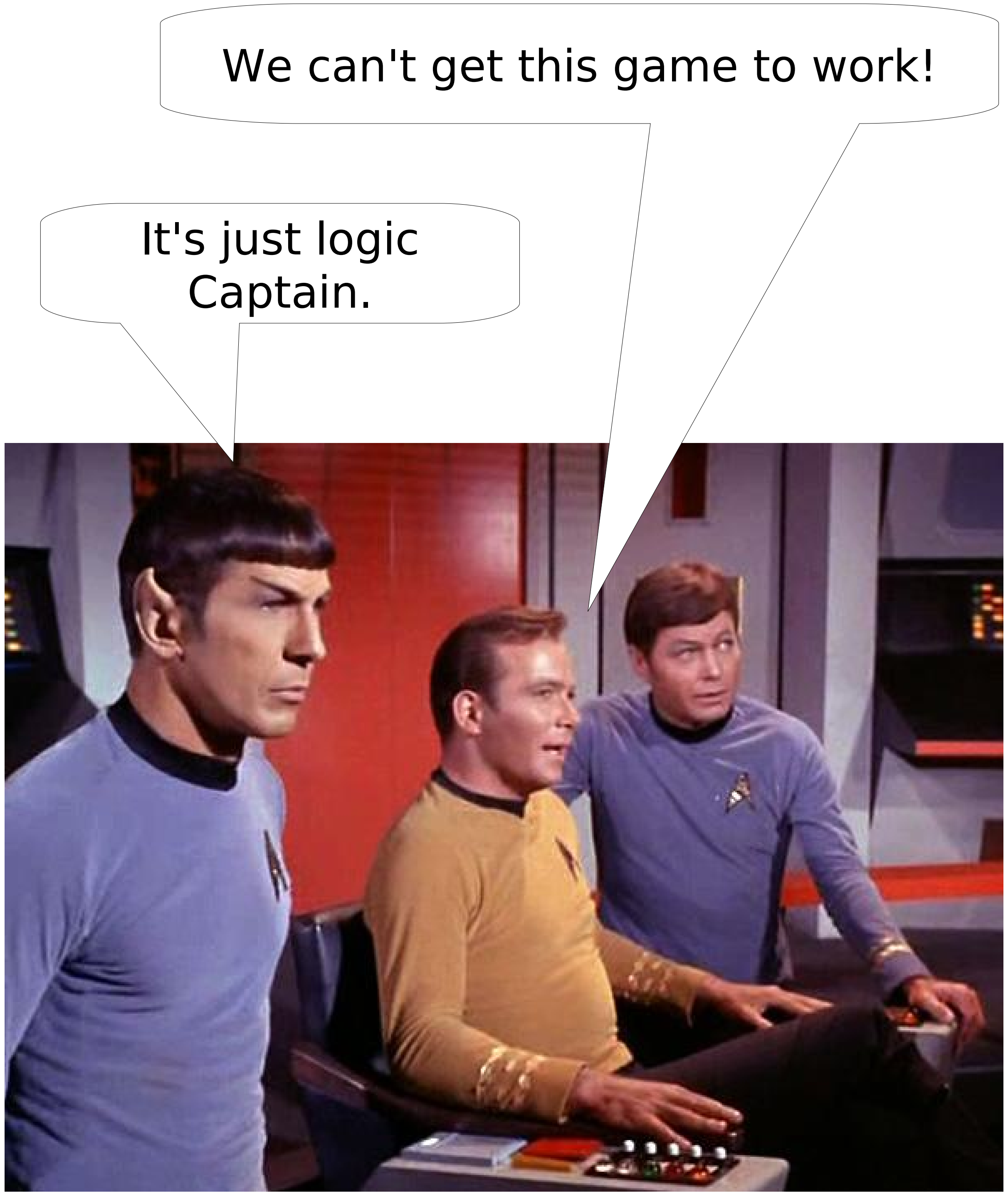


Recap So Far


loggedIn(..)
receivedMessage(..)
disconnected(..)
receivedMessage(..)
AppListener
ClientSessionListener
ChannelListener
AppContext
ChannelManager
TaskManager
DataManager
Project Darkstar Core Services

Recap So Far

- Network Communications handled automatically
- Multi-client communications natively supported
- Thread management and data consistency is transparent
- Persistence is automatic
- Supports scalable deployments with minimal additional effort
- Allows developers to focus solely on game logic

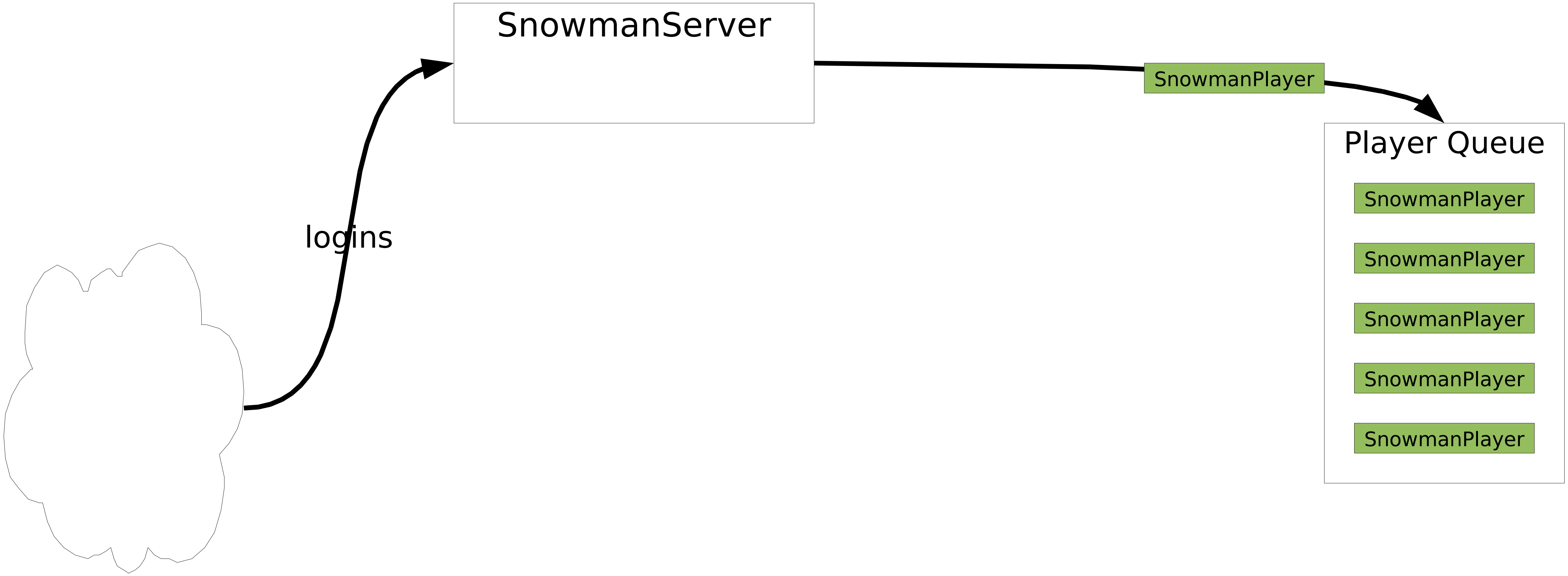


(Image From Star Trek: TOS *This Side of Paradise*)

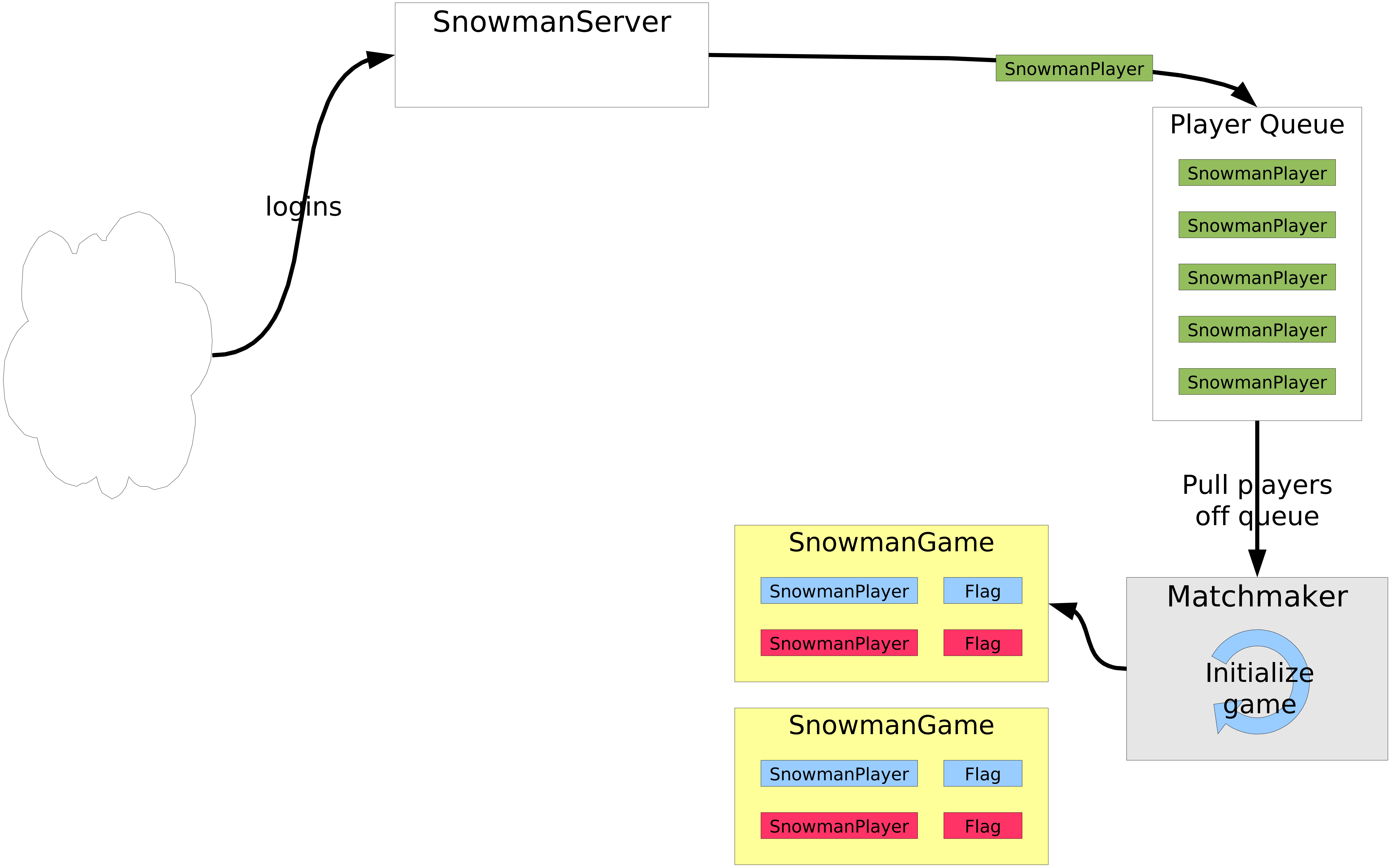
Example: Project Snowman

- Capture the Flag style snowball fight
- Rules:
  - Two teams of Snowmen players
  - Object of game is to retrieve opponents flag and bring it back to your base
  - Snowmen can throw snowballs at eachother
  - If a snowman gets hit with a snowball, snowman gets bigger and slower, but increases its attack range
  - After so many hits, a snowman will fall over, drop the flag (if holding it), and respawn

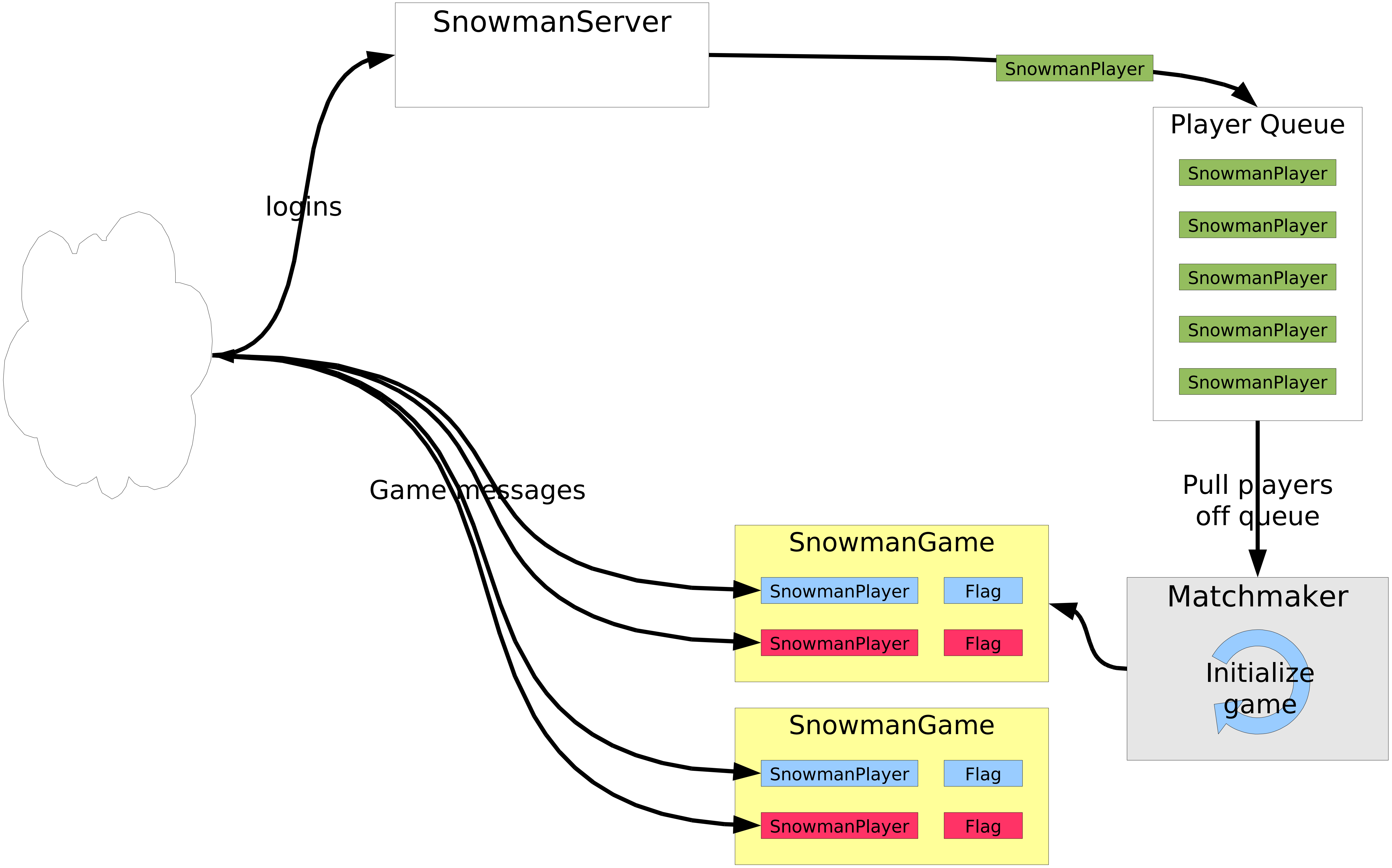
Project Darkstar



Project Darkstar

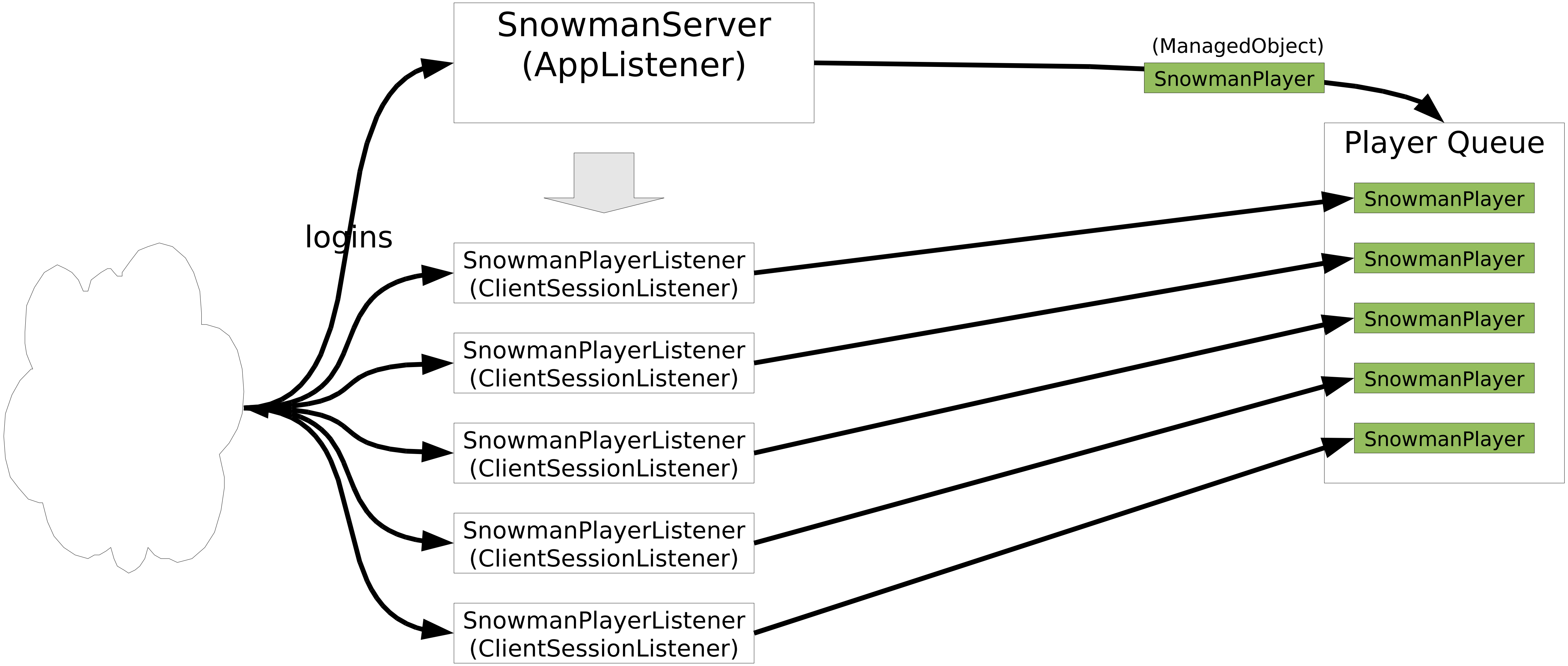


Project Darkstar



Developing Games With Open Source Technologies

Project Darkstar



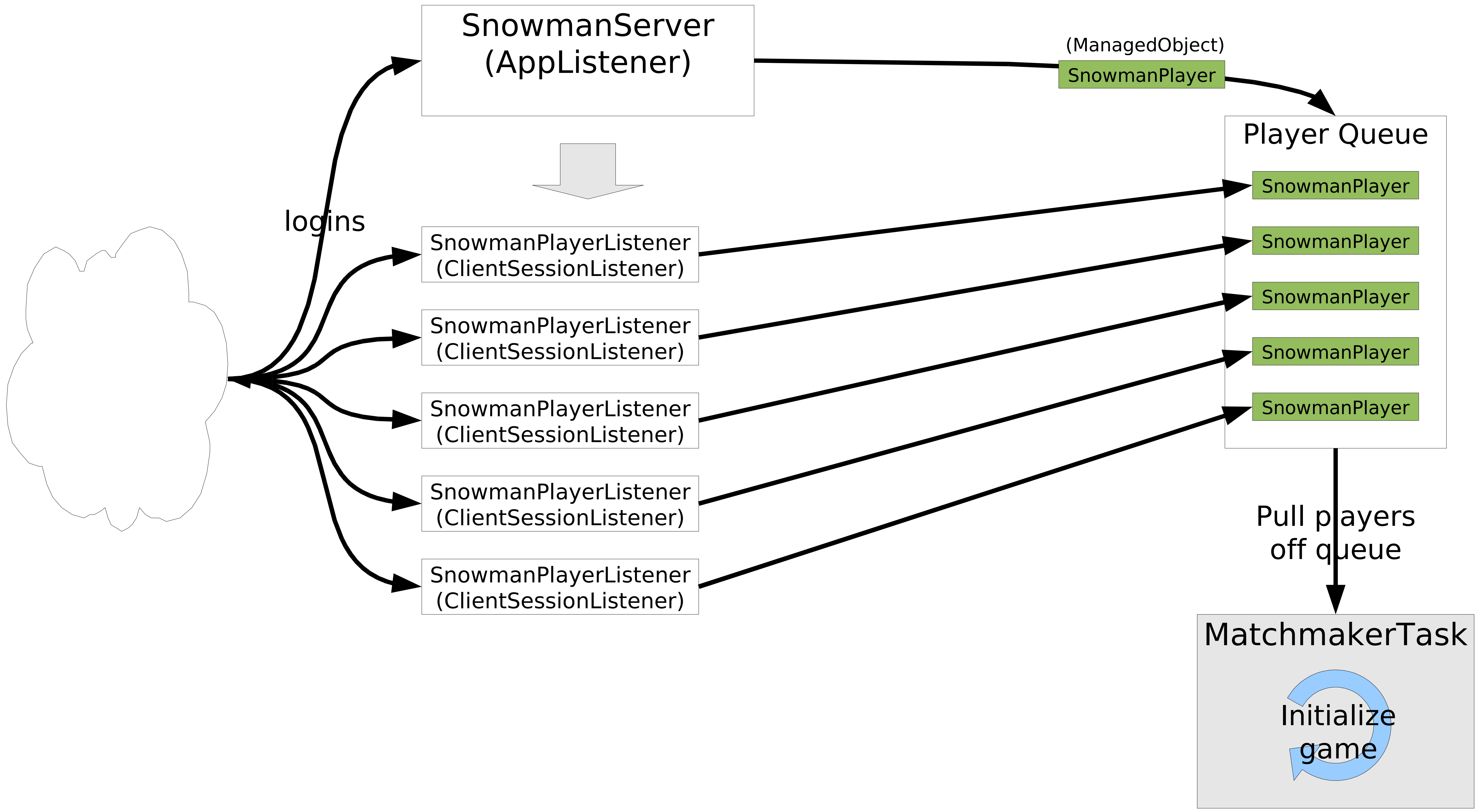
SnowmanServer Pseudo-code

```
public class SnowmanServer implements AppListener, ManagedObject, Serializable {
   private ManagedReference<Queue<ManagedReference<SnowmanPlayer>>> queueRef;
   ...
   public void initialize(Properties props) {
      ...
      //create the waiting player queue
      Queue<ManagedReference<SnowmanPlayer>> queue = INITIALIZE QUEUE;
      //save the queue into the data store by creating a reference
      queueRef = AppContext.getDataManager().createReference(queue);
      ...
      //create self scheduling MatchmakerTask
      AppContext.getTaskManager().scheduleTask(new MatchmakerTask(.., queueRef, ..));
   }

   public ClientSessionListener loggedIn(ClientSession session) {
      //create the player
      SnowmanPlayerListener playerListener =
        new SnowmanPlayerListener(.., session, ..);

      //retrieve the queue from the data store and add the player
      queueRef.get().add(playerListener.getPlayerRef());

      return playerListener;
   }
}
```

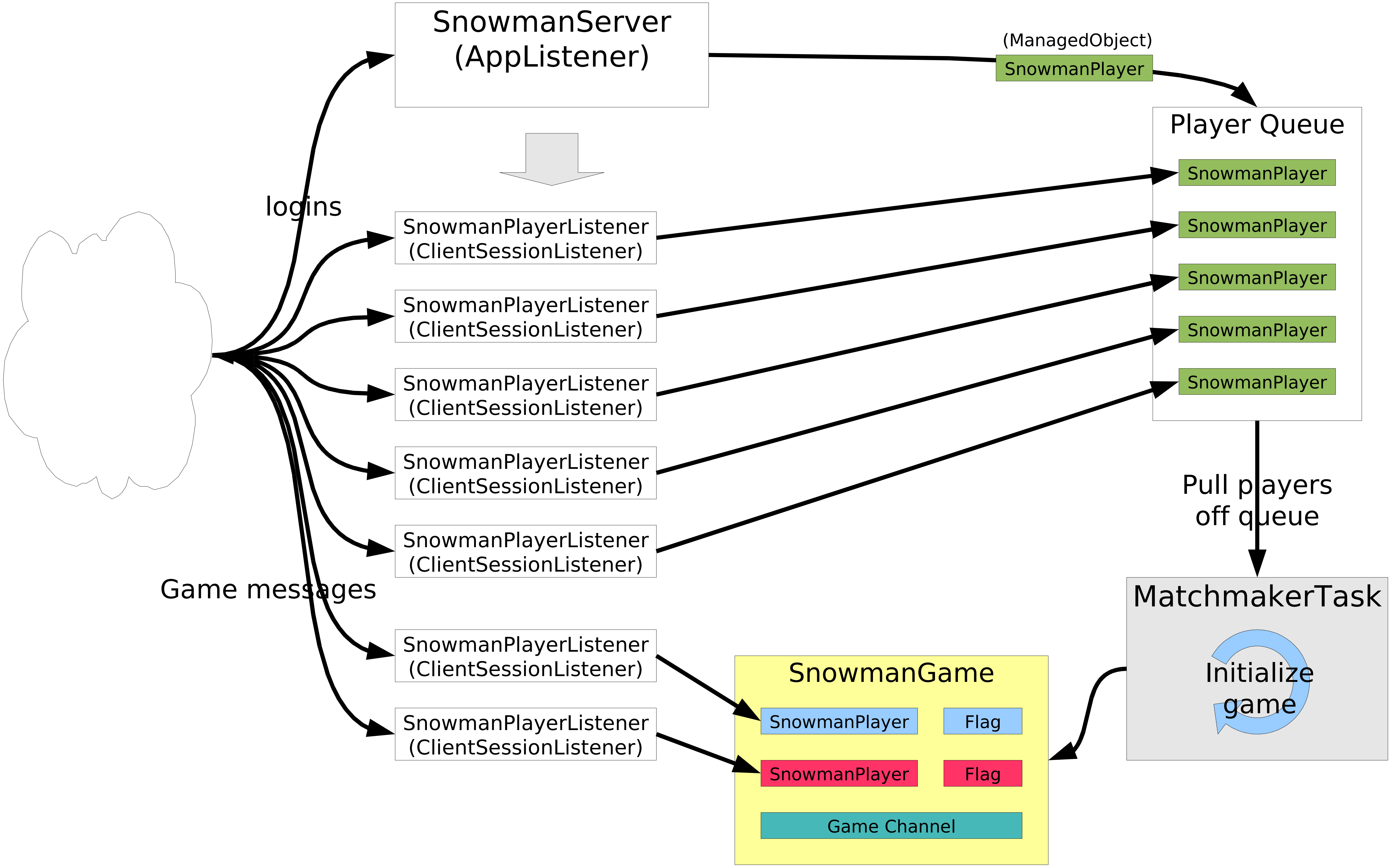
SnowmanPlayerListener Pseudo-code

```
public class SnowmanPlayerListener implements ClientSessionListener, Serializable {
   ...
   private final ManagedReference<SnowmanPlayer> playerRef;
   ...
   public void receivedMessage(ByteBuffer message) {
      //retrieve the player from the data store
      SnowmanPlayer player = playerRef.get();

      PROCESSMESSAGE(player, message);
   }

   public void disconnected(boolean graceful) {
      try {
         //retrieve the player from the data store for updating
         SnowmanPlayer player = playerRef.getForUpdate();

         if (player.getGame() != null)
                player.getGame().removePlayer(player);

         //remove the player from the data store
         AppContext.getDataManager().removeObject(player);
      } catch (ObjectNotFoundException alreadyDisconnected) {
         HANDLE EXCEPTION;
      }
   }
}
```

Project Darkstar



Project Darkstar



SnowmanServer (AppListener)

(ManagedObject) SnowmanPlayer

Player Queue

SnowmanPlayer

SnowmanPlayer

SnowmanPlayer

SnowmanPlayer

SnowmanPlayer

logins

SnowmanPlayerListener (ClientSessionListener)

SnowmanPlayerListener (ClientSessionListener)

SnowmanPlayerListener (ClientSessionListener)

SnowmanPlayerListener (ClientSessionListener)

SnowmanPlayerListener (ClientSessionListener)

Pull players off queue

Game messages

SnowmanPlayerListener (ClientSessionListener)

SnowmanPlayerListener (ClientSessionListener)

MatchmakerTask

Initialize game

SnowmanGame

SnowmanPlayer

Flag

SnowmanPlayer

Flag

Game Channel

MatchmakerTask Pseudo-code

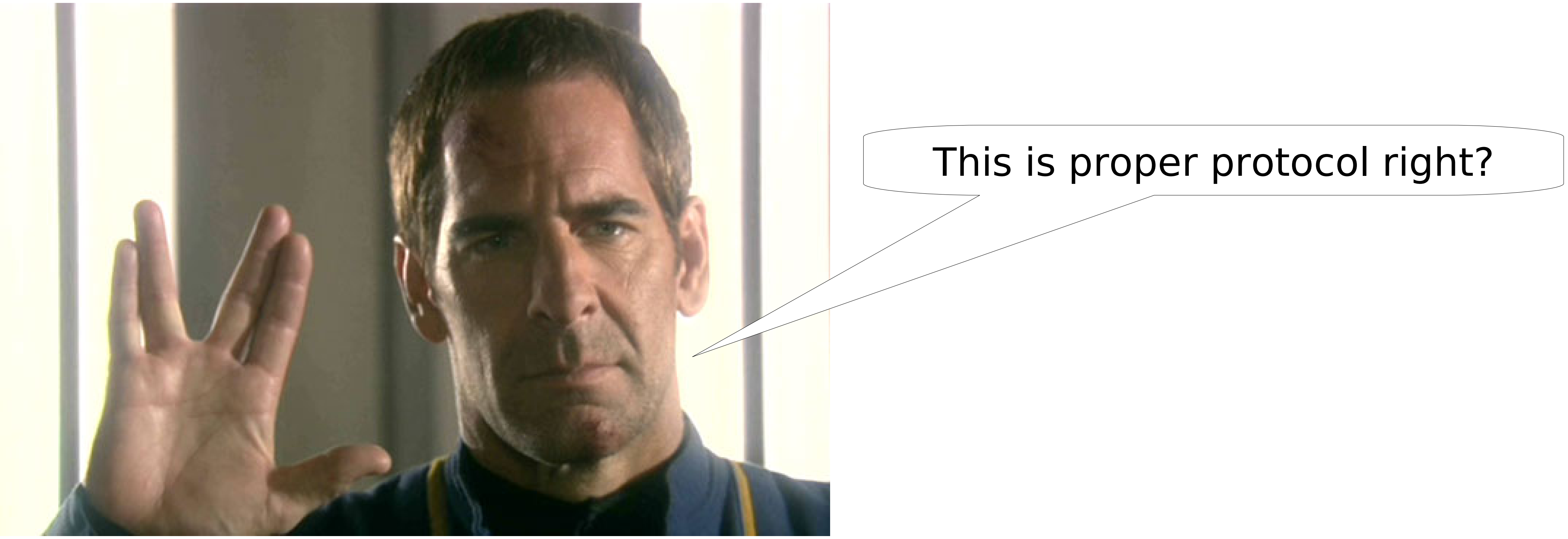
```
public class MatchmakerTask implements Task, Serializable {
   private List<ManagedReference<SnowmanPlayer>> waitingPlayers;
   private ManagedReference<Queue<ManagedReference<SnowmanPlayer>>> queueRef;
   ...
   public void run() throws Exception {
      boolean playersFound = false;
      for(int i = 0; i < numPlayersPerGame; i++) {
         //pull players off of queue
         ManagedReference<SnowmanPlayer> nextPlayer = queueRef.get().poll();
         if(nextPlayer != null) {
            playersFound = true;
            waitingPlayers.add(nextPlayer);
         }
         if(waitingPlayers.size() == numPlayersPerGame) {
            startGame(waitingPlayers); //create game with players and add to data store
            break;
         }
      }

      //reschedule task for the next cycle
      if(playersFound)
         AppContext.getTaskManager().scheduleTask(this);
      else
         AppContext.getTaskManager().scheduleTask(this, POLLINGINTERVAL);
   }
}
```

Project Snowman: Status

- Complete login and matchmaking system with minimal effort
- What's next?

Project Snowman: Message Protocol

- We need to define what messages the client can send, what messages the server can send, and how each message is processed



(Image From Star Trek: ENT *Kir'Shara*)

Project Snowman: Message Protocol

- We need to define what messages the client can send, what messages the server can send, and how each message is processed
- Project Darkstar allows us to think about this, and this alone:
  - No network communications code
  - No thread management
  - No synchronization required during message processing

Project Snowman: Message Protocol

- Each message is delivered as a ByteBuffer. The first byte in the buffer represents the message type. The remaining payload and number of bytes are determined by the message type and parsed out accordingly.

Client Messages

```
TYPE (payload)
 - Description...

MOVEME (float startX, float startY, float endX, float endY)
 - A MOVEME message is sent by the client with its current believed start
   position and its intended target move position

ATTACK (int targetId, float x, float y)
 - An ATTACK message is sent by the client with its intended target
   snowman id and its current believed position

GETFLAG (int flagId, float x, float y)
 - A GETFLAG message is sent by the client with its intended target flag
   id and its current believed position

SCORE (float x, float y)
 - A SCORE message is sent by the client its current believed position
```

Project Snowman: Message Protocol

- Each message is delivered as a ByteBuffer. The first byte in the buffer represents the message type. The remaining payload and number of bytes are determined by the message type and parsed out accordingly.

Server Messages

```
NEWGAME (int myId, String mapName)
STARTGAME ()
ENDGAME (enum endState)
ADDMOB (int id, float x, float y, enum type, enum team)
REMOVEMOB (int id)
MOVEMOB (int id, float startX, float startY, float endX, float endY)
STOPMOB (int id, float x, float y)
ATTACHOBJ (int sourceId, int targetId)
ATTACKED (int sourceId, int targetId, int hp)
RESPAWN (int id, float x, float y)
```

Project Snowman: Message Protocol

- Each message is delivered as a ByteBuffer. The first byte in the buffer represents the message type. The remaining payload and number of bytes are determined by the message type and parsed out accordingly.

Common Messages

`READY ()`

Example Message Processor: GETFLAG

```
public class SnowmanPlayerListener implements ClientSessionListener, Serializable {
   ...
   private final ManagedReference<SnowmanPlayer> playerRef;
   ...
   public void receivedMessage(ByteBuffer message) {
      //retrieve the player from the data store
      SnowmanPlayer player = playerRef.get();

      PROCESSMESSAGE(player, message);
   }

   public void disconnected(boolean graceful) {
      try {
         //retrieve the player from the data store for updating
         SnowmanPlayer player = playerRef.getForUpdate();

         if (player.getGame() != null)
                player.getGame().removePlayer(player);

         //remove the player from the data store
         AppContext.getDataManager().removeObject(player);
      } catch (ObjectNotFoundException alreadyDisconnected) {
         HANDLE EXCEPTION;
      }
   }
}
```

Example Message Processor: GETFLAG

```
public class SnowmanPlaye
   ...
   private final ManagedR
   ...
   public void receivedMe
      //retrieve the play
      SnowmanPlayer playe

      PROCESSMESSAGE(play
   }

   public void disconnect
      try {
         //retrieve the p
         SnowmanPlayer pl

         if (player.getGa
                player.ge

         //remove the pla
         AppContext.getDa
      } catch (ObjectNotF
         HANDLE EXCEPTION;
      }
   }
}
```

- ➔ Application code parses incoming message (ByteBuffer)
- ➔ Recognizes message as a GETFLAG message
- ➔ Extracts expected parameters from message (flagId, x, y)
- ➔ Calls `getFlag(long now, int flagId, float x, float y)` on the player ManagedObject

Example Message Processor: GETFLAG

- When a GETFLAG message is received, the `getFlag(long now, int flagId, float x, float y)` method is called on the associated `SnowmanPlayer` object.

- GETFLAG game logic:

```
IF SnowmanPlayer is DEAD
   NO-OP

GET the flag with id flagId.
IF there is no flag OR
   the flag is my team color OR
   the flag is already held
   NO-OP

IF x,y is a valid start position AND
   x,y is within grab range of the flag
   STOP movement of the SnowmanPlayer
   GRAB the flag
   NOTIFY all other clients with
    ATTACHOBJ message
```

Example Message Processor: GETFLAG

```
protected void getFlag(long now, int flagID, float x, float y) {
   IF SnowmanPlayer is DEAD
      NO-OP

   GET the flag with id flagId.
   IF there is no flag OR
      the flag is my team color OR
      the flag is already held
      NO-OP



   IF x,y is a valid start position AND
      x,y is within grab range of the flag
      STOP movement of the SnowmanPlayer
      GRAB the flag
      NOTIFY all other clients with
       ATTACHOBJ message
}
```

Example Message Processor: GETFLAG

```
protected void getFlag(long now, int flagID, float x, float y) {
   if(state == PlayerState.DEAD)
      return;

   GET the flag with id flagId.
   IF there is no flag OR
      the flag is my team color OR
      the flag is already held
      NO-OP



   IF x,y is a valid start position AND
      x,y is within grab range of the flag
      STOP movement of the SnowmanPlayer
      GRAB the flag
      NOTIFY all other clients with
       ATTACHOBJ message
}
```

Example Message Processor: GETFLAG

```
protected void getFlag(long now, int flagID, float x, float y) {
   if(state == PlayerState.DEAD)
      return;

   SnowmanFlag flag = gameRef.get().getFlag(flagID);
   if(flag == null || flag.getTeamColor() == teamColor ||
      flag.isHeld() || holdingFlagRef != null)
      return;



   IF x,y is a valid start position AND
      x,y is within grab range of the flag
      STOP movement of the SnowmanPlayer
      GRAB the flag
      NOTIFY all other clients with
       ATTACHOBJ message
}
```

Example Message Processor: GETFLAG

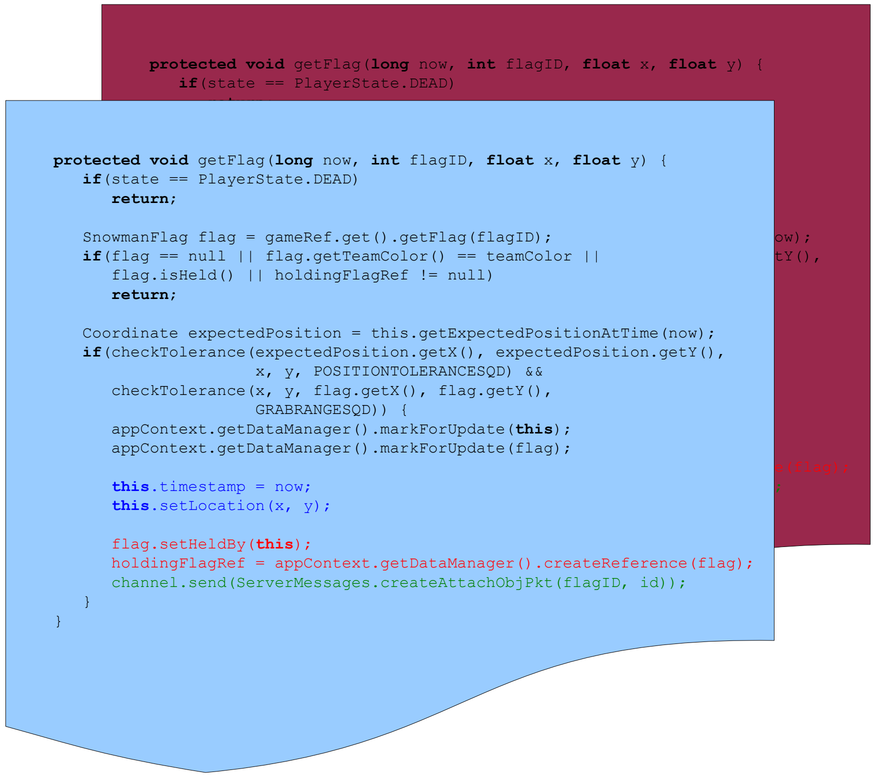
```
protected void getFlag(long now, int flagID, float x, float y) {
    if(state == PlayerState.DEAD)
        return;

    SnowmanFlag flag = gameRef.get().getFlag(flagID);
    if(flag == null || flag.getTeamColor() == teamColor ||
        flag.isHeld() || holdingFlagRef != null)
        return;

    Coordinate expectedPosition = this.getExpectedPositionAtTime(now);
    if(checkTolerance(expectedPosition.getX(), expectedPosition.getY(),
                        x, y, POSITIONTOLERANCESQD) &&
        checkTolerance(x, y, flag.getX(), flag.getY(),
                        GRABRANGESQD)) {
        STOP movement of the SnowmanPlayer
        GRAB the flag
        NOTIFY all other clients with
         ATTACHOBJ message
    }
}
```

Example Message Processor: GETFLAG

```
protected void getFlag(long now, int flagID, float x, float y) {
   if(state == PlayerState.DEAD)
      return;

   SnowmanFlag flag = gameRef.get().getFlag(flagID);
   if(flag == null || flag.getTeamColor() == teamColor ||
      flag.isHeld() || holdingFlagRef != null)
      return;

   Coordinate expectedPosition = this.getExpectedPositionAtTime(now);
   if(checkTolerance(expectedPosition.getX(), expectedPosition.getY(),
                     x, y, POSITIONTOLERANCESQD) &&
      checkTolerance(x, y, flag.getX(), flag.getY(),
                     GRABRANGESQD)) {
      appContext.getDataManager().markForUpdate(this);
      appContext.getDataManager().markForUpdate(flag);

      this.timestamp = now;
      this.setLocation(x, y);

      flag.setHeldBy(this);
      holdingFlagRef = AppContext.getDataManager().createReference(flag);
      channel.send(ServerMessages.createAttachObjPkt(flagID, id));
   }
}
```

Scenario: GETFLAG (Multiple Players)

- Contention handled automatically
- Object locking and transaction rollback/retry are transparent



```
protected void getFlag(long now, int flagID, float x, float y) {
   if(state == PlayerState.DEAD)
```

```
protected void getFlag(long now, int flagID, float x, float y) {
   if(state == PlayerState.DEAD)
      return;

   SnowmanFlag flag = gameRef.get().getFlag(flagID);
   if(flag == null || flag.getTeamColor() == teamColor ||
      flag.isHeld() || holdingFlagRef != null)
      return;

   Coordinate expectedPosition = this.getExpectedPositionAtTime(now);
   if(checkTolerance(expectedPosition.getX(), expectedPosition.getY(),
                     x, y, POSITIONTOLERANCESQD) &&
      checkTolerance(x, y, flag.getX(), flag.getY(),
                     GRABRANGESQD)) {
      appContext.getDataManager().markForUpdate(this);
      appContext.getDataManager().markForUpdate(flag);

      this.timestamp = now;
      this.setLocation(x, y);

      flag.setHeldBy(this);
      holdingFlagRef = appContext.getDataManager().createReference(flag);
      channel.send(ServerMessages.createAttachObjPkt(flagID, id));
   }
}
```

Writing a game with Project Darkstar

1. Design game world state with POJOs appropriately using **`ManagedObject`** and **`ManagedReference`**

2. Handle logins and disconnects by implementing appropriate API methods

3. Define message protocol for your game

4. Implement message parsing behavior

5. Implement handlers for each message, interacting with Project Darkstar's Manager objects

6. That's it!

Final Recap

- Project Darkstar allows developers to focus almost exclusively on game logic
- Strips away mechanics of burdensome requirements
    - Communications
    - Thread management
    - Contention management
    - Persistence
    - Automatic Scaling

(Image From *Star Trek: Generations*)

Project Snowman: Demo

- Checkout a live, playable demo of the game at booth 422
- http://www.projectdarkstar.com
- http://project-snowman.dev.java.net