Project Slipstream

An exercise in scalable design patterns and programming best practices for Project Darkstar

November 3, 2009 [draft v1]

0. Introduction

Slipstream[1] is an experiment in forming patterns and best-practices for programming to Project Darkstar. We wanted to capture experience from writing applications to share with the development community. More than just codifying patterns and giving examples, however, we wanted to create a framework: code that serves not only as tutorial but as a framework for game development.

The result is not a full "game engine" per se, but a good starting point to help developers focus on how their game logic could be designed. In effect, it herds development in the right direction for creating code that will scale well. It also uses specific interfaces like Game, Player or Region to give more clear direction about how to use Project Darkstar.

The motivation for this project comes from observing some common questions and confusion we often hear from first-time Project Darkstar users. In some cases Slipstream gives examples of how to use the server APIs. In other cases, it abstracts them away completely. Ultimately, the goal is to make it extremely easy to write a first game, and then start to learn more about the details and design of Project Darkstar as needed.

Slipstream is still in the early stages. There are a number of pieces missing, and some of these are discussed in the final section. From early experience actually writing game code using this framework, however, we feel confident that what exists today will be useful, and may encourage developers to contribute to and extend Slipstream to make it even more useful. If you're interested, please join the project! Details are in the final section.

In its current form, Slipstream is more focused on the server, and specifically basic game logic running in a Project Darkstar server node. This means that there is nothing for handling graphics on the client, AI or Physics modeling, etc. There are no extra management utilities and there is no integration with external services. Again, these would be great ways for you to contribute to the project, but were not the initial focus.

---

1 http://slipstream.dev.java.net

The rest of this guide is broken into a few sections. The first talks about some of the general advantages and challenges of programming to Project Darkstar, and what that means in terms of low-level best practices and higher-level scalable game design. The second section discusses how these experiences are applied in Slipstream, walking through the design of the core code and some example games built to this model. The third section demonstrates how custom Services can be used to abstract out some common components. The final section concludes this guide with some missing pieces and suggestions for next steps.

This guide is, in effect, a distillation of many distinct experiences, discussions and other guides. Without trying to name every source that's available, and with apologies to those we've missed, here are some other places to look for experience about how to use Project Darkstar:

- The community home for the core work: http://projectdarkstar.com
  - From the “Discussion” tab follow the links to our forums and blog roll
  - From the “Downloads” tab grab the latest stable releases
- The development home for the core code: http://games-darkstar.dev.java.net
  - Look here for the source code, open issues, and developer mailing lists
- The stable releases contain a tutorial and examples
  - Look in the doc directory for the tutorial[2], API documentation[3], and examples
- The example games like Snowman, DarkChat and Hack
  - Runnable games, available from the community home site

Hopefully this collection of documents and resources will help you get started or dive deeper into using Project Darkstar!

1. Effective Programming with Project Darkstar

Before understanding what Slipstream is and how it works, it's pretty important to understand some of the details of Project Darkstar, and how it can be used or abused effectively. If you just want to get started writing against the Slipstream APIs you can skip this section, but you may not fully understand all of the design you see. You have been warned.

Project Darkstar provides powerful features in a fairly small API. Many things are taken care of for you the developer. The tradeoff here is that you need to think about how you write your code in a slightly different way than you may be used to. The result is that some subtle details can make a significant difference in how well your code performs and scales.

This section starts with an overview of what some of these new programming approaches are, and why they make a difference. Next, this background is used to suggest some common low-level patterns and best-practices to keep in mind when writing code for Project Darkstar applications. Finally, these rules are applied to describe some common programming designs to consider for your application code as a whole. Collectively, these should help your applications

2 Available online at http://projectdarkstar.com/distributions/current/sgs-server/tutorial/ServerAppTutorial.pdf
3 Available online at http://projectdarkstar.com/distributions/current/sgs-server/doc/

run and scale better, and serve as the background for the next section which details Slipstream.

Note that this section is not an introduction to the Project Darkstar API, nor does it provide any examples of how to get started writing application code. It's assumed that you're already familiar with the basic interfaces, and would like to understand better how to use those interfaces effectively. If you aren't familiar with the application APIs, now would be a good time to go look at the tutorial and javadoc referenced above.

I. *Differences in the Project Darkstar model*

Project Darkstar provides a simplified view of a cluster. The illusion is that of a single system with single-threaded events. As an application developer, you never see when a node is added or fails and you don't need to think about how to exploit multiple cores or threads. The system appears to be available at all times, with no changes needed to application code wether you have one or hundreds of application nodes.

This model is supported by using an integrated transaction model. All events are handled as a separate task, and all tasks in an application are fully transactional which means that consistency is maintained between tasks. If there is conflict, one of the conflicting tasks is rolled back and re-tried. Only when a task succeeds are there any side-effects, meaning that no other tasks are scheduled, messages sent or objects updated if the task fails. When a task succeeds (i.e., commits), any changes made are updated automatically in the data store. In the database world, this is known as supporting ACID semantics[4].

This model has a lot of nice benefits to the developer. Use of transactions means that there is no explicit locking or synchronization needed which simplifies application code. Not having to worry about available resources and handling failure means that developers can focus on game logic. The trade-off for this model is that developers do need to focus on a different problem: contention.

One of, if not the main design issue that will determine how well an application scales with Project Darkstar is how much contention occurs. Essentially, if there are two tasks that both need to modify an object, or one task that wants to read an object that the other task is modifying, only one of these tasks can succeed. The other will have to abort and start over. The more times this happens, the more wasted work will occur in the system and the more tasks will be delayed.

As a developer, you need to think about these “hot-spots” in your code. In order for an application to scale, object access and sharing should be spread out as much as possible. This needs to be balanced, however, against the cost of fetching each object from the data store. There is a cost associated with each object access, and so application code needs to spread out possible conflict while minimizing the number of objects accessed in a task.

In addition to the cost of accessing the object, developers also need to think about how long a task runs. Obviously tasks need to be short to minimize latency observed by the clients. Tasks also need to be short to minimize conflict, since the longer a task runs the more likely it will

4 Note that the current work on a caching data store involves loosening some of the durability guarantees, but will still provide consistency and atomicity as expected. More details are available at the project web site.

overlap with another task that conflicts over some data access. To help with this model all transactions have a timeout[5] that will cause the task to be aborted if it takes too long. This means that the real design goals for a scalable application are to minimize the length of a task, the number of objects accessed in a task, and the possible points of conflict over object updates while still doing whatever server logic is needed to support the application.

The tool that application developers are given to help with this task is called a `ManagedObject`. This interface defines an object in the data sore that can be accessed by any task. It also defines "cut-points" in the serialization graph, so that developers can think about what data is being loaded with each object access and how to spread out concurrent modifications. This is a pretty simple interface but has some subtle effects on how an application scales.

The rest of this section is a collection of rules aimed at helping application developers write more scalable, correct code. This is by no means an exhaustive list. Instead, it's a basic guide for how to think about and design around the model discussed above. The next section will show these rules put to practice in the Slipstream codebase.

II. *Programming best practices and common issues*

- **Minimize conflict**

  As an application developer, you won't be able to avoid all conflict. By definition, in a transaction-based system some conflict will occur. Your goal is to minimize the conflict as best as possible. Think about the minimum set of objects you need for a given task, and the minimum state you need in each of these objects. If something doesn't need to be shared, don't share it. If something does need to be shared, see if it can be decomposed further, and make sure that it is modified infrequently if at all. Most of the suggestions that follow are suggested in this spirit.

- **Localize modifications**

  Try to keep your modifications as local as possible. In other words, think about the number of tasks that are likely to be interacting with the object you're modifying. Can you keep the object that's being modified local to a small area of your code or set of tasks? Consider the obvious example of a map from name to player data, where player data needs to be updated. The map can contain all of the data directly, which requires updating the map for any player change:

  ```
  PlayerData data = mapRef.getForUpdate().get("playerName");
  ```

  The map could also contain references to the player data, where each individual object is modified:

  ```
  PlayerData data = mapRef.get().get("playerName").getForUpdate();
  ```

5 The default timeout for a transaction is 100 milliseconds, but in practice tasks should never come close to running this long. See below for more details.

In the first case, every player update will contend on a single object. In the second, only updates to the same player will conflict.

Perhaps a more concrete example comes up when thinking about how physical world regions can be represented. Rather than having one monolithic data structure that has to be updated with each movement, or any time a Player joins or leaves the region, break the region down into sub-sections. This way two players can move at the same time, and as long as they're not moving to the same place, both movements can be handled concurrently.

- **Consider ManagedObject versus Serializable**

  In Project Darkstar you often have the choice of implementing `ManagedObject` on classes that must implement `Serializable`. Examples in the API are `Task` or `AppListener`, but this will be a general design decision in your own code as well. The implications of this decision are sometimes subtle. Generally, any object that needs to be shared between other objects must implement `ManagedObject`. Because each `ManagedObject` access requires a separate data store call, it's preferable to limit the number of references used. If some state is private to a given object and accessed frequently, it's almost always better not to implement `ManagedObject` for this state (the proxies in the next section are a great example of this)

  In addition to the number of accesses, it's important to think about about the size of the objects. Each object access requires deseriailzing that object, so if the object is large (or could grow large, as in the case of `Collection`) then this will be expensive. In that case it might make more sense to break the object into smaller objects, especially if not all the state is needed in every task.

  There are many interfaces in Project Darkstar that accept instances of types that may optionally implement `ManagedObject`, and in most cases if you don't implement `ManagedObject` the server will take care of cleaning up these objects when they're not needed. This means that it usually makes code simpler if you don't implement `ManagedObject`. With these same methods, if `ManagedObject` is not implemented on the provided objects then this often results in fewer datastore accesses. It's not always the case, but generally it's preferable not to implement `ManagedObject` when it's not needed:

  ```
  public static class MyTask implements Task, Serializable {
      // …
  }

  // …

  AppContext.getTaskManager.scheduleTask(new MyTask());
  ```

  In this case, the new instance of MyTask will be removed from the data store once the task has run, and accessing the task instance requires only a single data store operation by the system. If MyTask implemented `ManagedObject`, then it would be up to the

developer to remove the instance once the task was no longer needed. Note also the comment below about marking updates for another aspect of implementing `ManagedObject`.

- **Make listeners immutable and Serializable**

  Generalizing from the previous point, whenever possible make the interfaces that you pass to Project Darkstar for purposes of getting called-back immutable, and do not implement `ManagedObject`. This applies not just to `Task`, but also to `AppListener`, `ClientSessionListener`, and `ChannelListener`. If these implementations don't get updated after they're constructed, and don't implement `ManagedObject`, then the system will generally perform better. As described above, this may also result in state being cleaned up correctly, whether its a `Task` being removed once run or a `ClientSessionListener` being removed automatically when the associated client disconnects.

- **Mark Updates, usually**

  Project Darkstar provides the ability to mark a `ManagedObject` when it is modified. This is optional, and is a way to tell the server that some object is being modified as part of a transaction. By default the server will look at all objects accessed during a transaction to make sure that it knows which ones were modified, so applications don't ever have to call the marking methods, but there are some times when it makes sense to do so.

  The `getForUpdate()` and `markForUpdate()` calls should only be made when it's clear that the object really is being updated. If it's not clear whether the object will be modified, it's preferable to start by calling `get()`, and later call `markForUpdate()` when it's clear the object is actually being updated. The marking itself should typically be part of the internal operation of an object, in other words it's usually preferable to `get()` an object and then let that object mark itself when it updates internal state:

```
public void setName(String name) {
    AppContext.getDataManager().markForUpdate(this);
    this.name = name;
}
```

  Note that it may not always be possible to mark an update. For example, only objects that implement `ManagedObject` can be marked for update. Sometimes this means choosing between a design that allows for marking and a design that optimizes general access. While it's hard to provide a concrete rule, generally the later (optimizing for general use) is preferred.

  It's hard to characterize whether aggressive marking helps or hurts an application, but generally if you've done a good job of minimizing conflict, then marking updates will help.

  One final note is that there is a logger that reports any un-marked modifications[6]. This

6 `com.sun.sgs.impl.service.data.DataServiceImpl.detect.modifications.level`

can be really useful if only as a tool to show you how your application is behaving. There's also a property to turn off update detection[7]. This will cause your application to break unless everything is marked correctly, so while it could help with performance it's generally discouraged behavior.

- **Prefer types to ManagedReference**

  When designing application interfaces, prefer passing around types instead of `ManagedReferences` to your types. In other words, try not to do this:

  ```
  public void addItem(ManagedReference<Item> itemRef) { /* ... */ }
  ```

  Instead, try to design your interfaces to look like this:

  ```
  public void addItem(Item item) { /* ... */ }
  ```

  It's fairly inexpensive to call `createReference()` if a given method needs to create a reference to the object. It will definitely help your application design, however, and long-term code maintenance if you follow this pattern.

- **Don't ever catch Exception**

  Whenever a task fails, whether due to conflict, timeout or some other internal error, an exception is thrown to halt the transaction. The transaction is marked as aborted, so it's guaranteed to fail even if the system doesn't see the exception, but if you catch `Exception` from a call that fails:

  ```
  ManagedReference<MyObj> objRef;
  // …
  try {
      // assume this call fails, causing the transaction to abort
      objRef.get().callMyMethod();
  } catch (Exception e) {
      // log the error and keep going
  }
  ```

  then the Transaction will keep running even though it can never commit. This means that the application will now be doing wasted work, and the final exceptions that are logged may make it much harder to find the root failure. Limit your exception handling to the exceptions that you know your code should be reacting to.

- **Use Serial Version**

  Not a Darkstar rule per se, but make sure to provide a version identifier for all Serializable types:

---

7 `com.sun.sgs.impl.service.data.DataServiceImpl.detect.modifications`

```
private static final long serialVersionUID = 1;
```

This will help in the long-run with managing changes and updates.

- **Avoid static, synchronized and transient**

  A hard rule in Project Darkstar is that application code must never do any explicit synchronization or locking. Generally, application code also shouldn't try to use any thread or machine local code. This means that the `synchronized` keyword should never appear, and no concurrent data structures or thread locals should ever be used. It also means that `static` should almost never appear in application code.

  There are two notable exceptions for the use of the `static` keyword. The first is for defining constants, which is always a good programming practice:

```
public static final String MY_OBJ_NAME = "some.constant.name";
```

  The second is for member classes. Because any non-static inner classes have an implicit reference back to their parent, this usually results in violating the the serialization rules in Project Darkstar. So, inner classes should always be static:

```
public class MyClass implements Serializable, ManagedObject {
    // …
    public static class MyTask implements Task, Serializable {
        // …
    }
}
```

  Care should be taken when using `transient` variables, because each transaction results in newly deserialized versions of objects. Unless you have a very specific reason, you should avoid using `transient` variables.

III. *Application design patterns and scaling considerations*

- **Use separate instances for any listener**

  Generally, any listener or callback interface should be a separate instance. This means that an application usually shouldn't re-use a `ClientSessionListener` instance for multiple `ClientSessions`, the same instance of `Task` for multiple simultaneous transactions, etc. This is mostly true because it can quickly create common points of conflict or confusion in design. As long the above rule about making these classes immutable is followed then this is less likely and this rule can be used more as advice.

  Along these lines, `ChannelListeners` should only be used if they're actually needed. If a `Channel` can go un-filtered, this will improve performance. For a `Channel` that only allows the Server to send messages, however, using a `ChannelListener` is generally ok since this overhead will only be incurred for handling messages from clients. Since clients shouldn't be sending messages, this should happen infrequently.

- **Decide when to use Sessions versus Channels**

  `Session`s allow direct messages between the client and server. `Channel`s allow messages to a group of any size, possibly filtered. A `Channel` could have only one member, thus mimicking a `Session`. While there are many ways to use `Session`s and `Channel`s, the best rule is to pick a design pattern and stick with it.

  Generally, a good rule is to use the `Session` interface for all direct messages between the client and the server. These messages are typically command messages, and scoped to the state associated with the client. For instance, a request to move, or drop an item, or cast a spell, or change some state will typically be a `Session` message. Likewise, if an illegal move is requested, or some state change happens private to a given client, this should be sent to the client through the `Session`.

  If the server wants to send a message of general interest, like changes to an open space, someone joining or leaving a game, news about the game, etc. then this should almost always be sent via a `Channel`. Things like group chat are also usually implemented with `Channel`s. Beyond this, it's up to the developer how else to use `Channel`s. Note that there will always be more overhead involved in sending to a `Channel` versus sending to a `Session`.

  `Channel`s have the nice property that they give context to a message. So, it might make sense to send messages to clients on `Channel`s only to give some extra detail about why the message is being sent, and to make it easier to handle the message on the client. This can be done in sending to the server too, but it will always result in exta overhead, so it's generally discouraged. As above, a developer will need to think about which `ChannelListener`s will be called to handle these messages, and what affect this may have on contention or data store accesses.

  Whatever design is chosen, it's important to remember that `Channel`s are designed for groups that don't change rapidly or have massive numbers of members. This means that you should never, for example, have a single `Channel` with every active player joined.

- **Keep private state local to listeners**

  In addition to preferring immutable listeners, if there is state that is private to that listener, or any associated users (e.g., the client associated with a `ClientSessionListener`), and it's used regularly by the listener, then it's almost always preferable to keep that state local to the listener. In other words, don't make it a separate `ManagedObject`. This generally results in fewer data store accesses and fewer opportunities to leak objects in the data store.

  An example of this is state in a `ClientSessionListener` needed to handle incoming messages. Another example is state in a `Task` that defines MOB behavior on the server. In these examples, if the state is needed each time these interfaces are called, but no other objects will need access to this state, make it immutable and keep it local to the

listener.

Note this is discussed above but called out more explicitly here because this is generally a very good rule to adhere to. Again, the proxies in the next section are a good example of this practice.

- **Prefer more, shorter tasks**

  Generally, using more asynchronous tasks that do a little work will perform much better than using a few, longer-running tasks. This means avoiding doing any unbounded work like iteration over a collection of an unknown size. Instead, spread out the iteration or work, possibly using the `shouldContinue()` routine if you need to do some large iteration or other unknown-size computation.

  This rule is especially true when dealing with large number of `Channels`. Generally, a given `Session` should not be a member of large numbers of `Channels`. This said, its always true that you should avoid joining/leaving a `Session` from large numbers of `Channels` within a single task. You should also avoid sending large messages, or spending a long time constructing complex messages.

  One common place where this is an issue is on logout. When a client disconnects, there's often a lot of state to clean up. Consider splitting up these updates into separate tasks, where any given task does some of the work and then schedules another task. Similarly, this pattern should be considered when initializing or destroying some large state associated with a game.

  Note that while the default transaction timeout is 100 milliseconds, in practice tasks should run much faster. Aim for tasks more on the order of 5 milliseconds in length. This will result in a much better-performing system. There is a property to change the timeout length[8], but developers should avoid any possible temptation to change this value.

- **Consider the scalable data structures**

  Project Darkstar includes implementations of some scalable data structures like `Set` and `Map`. These are designed to scale much more effectively than the standard classes provided in Java. If you only have a few elements you need to put into a collection, and if there's likely to be little or no possible conflict on a collection then it's ok to use the standard `Sets`, `Maps`, etc. from `java.util`. Otherwise, you should consider using the scalable classes found in `com.sun.sgs.app.util`.

  Part of what makes these scale well is that they handle concurrent access by splitting the internal structure into separate `ManagedObjects`. This also means that to access a given element, the whole structure doesn't need to be deserialized. The tradeoff here, obviously, is that it requires more data store accesses and could still result in conflict on the internal structure. This is why they tend to be more effective for larger data sets.

8 `com.sun.sgs.txn.timeout`

- **Use Profiling and Management while debugging**

  Project Darkstar includes a fairly rich set of interfaces for profiling a server and managing how it behaves. Between many tunable properties, `JMX` hooks and a custom profiling system, you can collect significant data and use that to tune the system. Don't wait until you're ready to deploy to look at this data. Consider profiling part of the debugging and design stage. Be proactive in understanding how your code is scaling, and where bottlenecks are forming. All of the hints and rules in this section have the common thread of helping to minimize concurrency delays, task failure and response time, but the only way to see how well you're doing this is by watching the live data your application generates.

2. Introduction to Slipstream

This section introduces Slipstream. The first part discusses the code itself, and how it captures some of the patterns discussed in the previous section. The second part walks through some examples of how a developer can use Slipstream to write a game.

I. *Slipstream: Design and Core Interfaces*

Slipstream is a simple, light-weight framework. The goal is to provide some core and utility interfaces that help developers focus their game design. This part covers the most important of these interfaces, and the design goals to explain why the code was structured this way. These interfaces may be useful directly, or as examples of how to write your own code.

- **Player**

  This is one of the basic elements of any game. While there are different kinds of players, generally a player is an active participant in a game. Examples would be a human's character, an AI-driven monster or NPC, etc. A Player has some name and unique identity, and usually some inventory. Because a Player takes an active role in the game where it is playing, you can send Messages to and between Players.

  Generally speaking, Players represent some persistent state. This state may last only for a limited scope, like a monster that exists for a single battle, or for an arbitrary time, like the state associated with a client's character. Still, this is data that needs to be available across the cluster, and may be accessed by many simultaneous tasks at once. Typically these accesses will be to observe state though obviously some state (health levels, temporary status, location, etc.) will need to change on a regular basis.

  When implementing a Player, it's best to think about which elements of the state are only needed or modified occasionally, and which need to be used on a regular basis. For instance, if some game requires that a Player update some of their Player statistics with nearly each move, then this suggests a value that should be directly included in the

Player's state. If there is a large collection of rarely accessed stats, then these could be stored as a separate `ManagedObject`.

- **UserPlayer**

  This is one of the two specific types of Player that are defined in Slipstream. A UserPlayer is a Player that is controlled by some connected client. This means that in addition to the usual details of a Player, this entity will have some connected status, will want to send and receive messages over the network, etc.

  One of the things that Slipstream hides in its APIs is the details associated with a connected `Session`. This helps simplify the various states that a game developer needs to keep track of. A UserPlayer looks just like any other Player from the developer's point of view, therefore, expect that it needs to maintain some state about what part of the game it's currently playing (see below).

  As a further utility, the BasicUserPlayer implementation is provided. This class provides common behavior to get a developer started. It leaves the developer with very little to define in order to get a UserPlayer supported. Typically this only needs to be extended further or overridden to support models like UserPlayers that can assume different characters within a given game. For most common uses, the BasicUserPlayer implementation will give developers the general functionality that they need.

- **Game**

  Another basic element of any game is the notion of a starting point. In Slipstream, this starting point is the Game interface. It is where all initial state is created, where the system looks up UserPlayer state when clients connect, and the central point that defines how UserPlayers move between different parts of the game.

  When a client connects, your Game will be asked for a UserPlayer instance, and then will be asked where to start the UserPlayer (see GameMode below for details). As a UserPlayer continues playing and moves between parts of the game, the Game interface is asked about how to handle these changes.

  The Game instance is a singleton, and there are, intentionally, no methods to ask for access to this object. Instead, it's provided to a few key methods that need to handle the basic lookup and coordination tasks described above. While this object can implement `ManagedObject`, it's best to make it only implement `Serializable`, and only keep as its direct state anything that will be needed on most or all interaction.

  Because this is a central point in the system, it's also best to make your Game implementation immutable. If does need mutable state, try to spread this out into separate `ManagedObjects`. Think about using name-bindings to your UserPlayers or elements of game-play to keep the Game instance itself as small as possible, since it will be read on a regular basis. To this end, there is a utility class provided to help keep track of UserPlayer objects.

- **GameMode**

  Within the scope of a single Game there are always many aspects, or modalities of game play. For instance, a player might be in a lobby, at a card table, participating in an auction, exploring an open world, etc. In Slipstream each of these is represented by a GameMode. These modes may or may not have some physical space associated with them.

  Abstractly, think about a GameMode as a set of rules or distinct aspect of your game. If you have towns that a Player can explore, there are probably rules about how Players interact in a town, what a Player is allowed to do, how they move around, etc. These rules are usually distinct from other aspects of game play. A game in Slipstream would probably define a single implementation of GameMode to support all towns but keep a different instance of that implementation for each town that a Player might visit.

  A given Player will only be active in one GameMode at a time. They may still be chatting with friends, or watching for updates in an auction, but their attention may only be focused on one modality of game play. In Slipstream, UserPlayers are always in some mode, and it's up to your Game logic to decide how that player transitions between modes. Each GameMode will define the rules of that part of the game but will also probably keep some state based on the active Players, past events, etc. This means that there will almost certainly be some immutable and some mutable state. Again, try to minimize the mutable state, and the frequency with this needs to change. If it's not changed regularly, keep it in a `ManagedObject` separate from the GameMode itself.

- **GameProxy**

  To help enforce how state is managed and who makes changes to that state, GameModes use a Proxy pattern to maintain per-Player state. Each Player that joins a GameMode gets a GameProxy instance to use when interacting with that mode. This is the interface that is actually used to handle Messages from the Player, and is a good place to store the state specific to the Player for the GameMode.

  Because these are typically private to a given Player, and because these proxies are usually needed any time a Player wants to interact with the game, implementations of GameProxy generally only implement Serializable. These become local state to the Player, and a simple way to delegate message-handling while minimizing possible sources of conflict. For the GameMode, it means that when a message needs to be handled (walk North, talk to townsfolk, take 2 cards, enter a new game, etc.) there is no need to do any extra lookup to find the Player state.

Before continuing with the Slipstream design, here is a quick summary of the components discussed so far. When Darkstar first starts up a Game is created which initializes state, probably including some number of GameModes. When a client connects to the Server, their UserPlayer object is requested from the Game, and this is joined to some GameMode by the Game. Now the client is actively participating in some aspect of the game. When the client sends a message to the server, this message is handled by the current GameProxy for that

user. The GameProxy is part of the UserPlayer state so no extra datastore lookup is needed and there will be no conflict accessing this state. The state associated with the user is available to the logic about how messages are handled in the current mode so the server can handle the message quickly. It's up to the GameMode, coordinated through its GameProxy, to determine how any given message is interpreted and what response if any is needed.

This covers most of the basic building blocks for the game logic and flow of client messages. The next thing to understand is how Players of different types interact, how messages are represented, and how different kinds of physical spaces affect this.

- **Region**

  As discussed above, some modes of play may not have any notion of geographic or spacial rules and interaction. In other words, they may not allow you to move around or change the way you play based on a position. Auction houses and shops are often presented as simple chat or browsing interfaces. Card games let you "pull up a chair at a table" but don't usually need any specific rules about moving around the space.

  For many online games, however, spacial rules are very important. Supporting this is the Region interface. A Region implementation defines a position-based space with a known set of rules about how movement occurs legally, how players interact within that space, etc. Players and Items (see below) may occupy a Region. Regions can also be hierarchical, to support special kinds of spaces within a larger space. Most methods for Regions use Coordinate, an abstract 3-space point.

  Probably the most obvious example of a Region is an open world. It has rules about where you can walk and what things (water, trees, mountains, buildings, etc.) block your progress. It will define how close you need to be to other Players to see or affect them, and how movement or status updates are broadcast to other members of the Region. It will know when you walk into some specific spaces (poisonous marshes, an exit to a town, a transport to another world, etc.) but not what the meaning of those spaces is.

  Typically a GameMode that needs some spacial representation will contain a reference to a Region. The GameMode is responsible for implementing the rules of game play, while the Region knows the rules about spacial movement and interaction. When a Player triggers some game-specific event (interacting with another Player, walking into some special space, etc.) the Region will communicate this event through the Player's GameProxy. Likewise, the GameMode will tell the Region when a Player is trying to move or interact with other Players, but will leave it to the Region to know who is in range, whether a given move is legal, etc. This keeps the game logic and play rules decoupled from the abstract behavior of a physical region.

  This separation is important because it means that a Region can be implemented completely separately from any game. This allows one developer to build, say, a mesh-based Region which any other game developer could use to support their GameMode. In addition to supporting good Darkstar design, the hope is that this could result in more reusable components and faster development.

- **RegionProxy**

  As with GameModes, Regions provide a proxy when some entity joins that Region. The RegionProxy is specific to that entity, and can be kept private to their state (as with GameProxy objects). The RegionProxy objects are what actually support requests for movement, leaving a Region, finding Players within range, etc.

  The design here is pretty much the same as with GameProxy. The idea is to create a small proxy that co-locates basic message-handling with the required user state. This is per-Player state that is typically only Serializable, though if you have a game where only a few messages result in region-based events, you may want to make the Proxy object separate from the Player state since it will be accessed infrequently.

- **Group**

  To this point the discussion has focused on individual Players and game logic, but obviously a significant reason for a game server is to allow group interaction. The Project Darkstar APIs provide `Channel`s as a way to allow many clients to communicate or the server to broadcast events to groups of clients. In Slipstream there is a Group abstraction above this.

  The Group interface is important for a number of reasons. Primarily, Slipstream defines different kinds of Players, any of whom may need to participate in group communication, and so a Group allows Players to join (not just clients). When you send something to a Group, you're actually sending a Message (see below), not encoded bytes. The message will be encoded for clients that need to hear the message, but will be kept in object form for the other Players. This makes it much easier to communicate to different kinds of Players and much more efficient to pass around messages.

  Like `Channel`s, Groups allow you join or leave a Player. Also like `Channel`s, a given Player shouldn't be a member of too many Groups at once, and shouldn't try to join or leave many Groups within a given task. In fact, most Groups will be supported by the abstract ChannelGroup class which uses a `Channel` for all UserPlayers in a Group, so keep in mind all the usual rules for `Channel`s when using Groups.

  While you are free to implement any kind of Group you want, some common ones are implemented for you that should cover most use-cases. These are UnmoderatedGroup (allowing anyone to send to all Group members), SingleSenderGroup (allowing only the server, or possibly one other Group member to send messages) and FilteredGroup. The latter defines a simple GroupFilter interface that can be used to decide on a per-message basis whether to allow the message or how it must be modified. Obviously this last type of group incurs more overhead, but is also more flexible.

  The UnmoderatedGroup will most often be useful for applications like Chat, or other types of group communications where you don't need to validate the messages or observe the messages at the server. SingleSenderGroups are useful for cases when the Server needs to broadcast updates or when a single Player has sole authority to

broadcast some state. This is what many Region implementations will use to handle localized groups that need to hear from the server about updates within range.

Groups do very little else by default, but they will handle membership updates if this is required. So, as a developer, you don't need to implement your own logic to let Players know when someone joins or leaves a Group. This is probably more important for chat groups, and may not be important (or even desired) for event broadcasts.

- **MOBPlayer**

  As was mentioned previously, there are two basic types of Player defined in the Slipstream APIs. UserPlayer has already been discussed. The second common type of Player is MOBPlayer. This is a general type to define any Player that doesn't have an associated client driving the Player behavior. Instead, there is some AI or other server-defined logic driving the behavior. Typical examples would be a Non-Playing Character (e.g., someone you talk with, who gives you quests), an AI-driven character who joins your party, or a monster. In Slipstream, any Player that isn't associated with a client is a MOBPlayer.

  A MOBPlayer starts out in a passive state, and needs to be started explicitly. Once started, it may be stopped and re-started as needed. Like all Players, MOBPlayers can react to messages, though unlike UserPlayers a MOBPlayer may decide not to subscribe to all messages (e.g., an NPC may not care about chat messages or movement updates in a Region).

  A MOBPlayer, in addition to reacting to messages, may also be proactive in its actions. This comes in the form of call-backs to the player's interface at regular (periodic) or irregular (MOB-defined lengths of time) intervals. A MOBPlayer can interact like any other Player in its current GameMode or Region, and can decide what it wants to do with each call-back or message.

  Unlike UserPlayers, MOBPlayers do not move between GameModes. Or, rather, there is nothing in the Slipstream interfaces to support this. The assumption is that a given MOB will typically exist only within a given aspect of play. Unlike UserPlayers, there will often be MOBPlayers that exist for only a short time.

  Regular and Irregular MOBPlayers are driven by scheduled tasks. These tasks use the `RunWithNewIdentity` annotation so that they they inherit a new identity, separate from the caller. Identity is a somewhat subtle issue in Darkstar, but the important detail here for application developers is using this annotation allows tasks to be better load-balanced and moved to machines with other tasks that need access to the same data. Essentially, if a given task represents the start of some new behavior for a new "persona" or entity, then you should use this annotation to help the system scale.

- **Message & Event**

  The final key building blocks for Slipstream applications are Messages and Events. Message is the generic abstraction for all communication between Players (on the server and between clients and the server), and from game logic to Players. Each Message communicates some specific, known detail. A client might send a Chat or Movement Message to the server, a Player could send an attack Message to another Player, or a GameMode could send an update to a Group. In all these cases, the game logic is separated from the details of message-encoding, or how the message needs to get communicated to individual players.

  This is important for a few reasons. For one, it means that server-side MOBPlayers don't have to decode byte-encoded messages to understand what they say, and the same object can be passed to many Players to communicate what's happening. For another, it means that the details of how Messages are encoded or decoded for communication with Clients can change without affecting the game logic. Finally, it makes it easier to evolve game logic or introduce new messages (as opposed to, say, defining each message a different known method on your interfaces).

  Some Messages may require no interpretation at the server. For instance, a Chat Message might just be delivered to an UnmoderatedGroup. On the other hand, a MovementMessage will clearly need to be handled by the UserPlayer's GameProxy. This may result in a response Message to the UserPlayer, or to other Players. Some of the more common Message types have an interface and implementation in a shared package for use by the client and server. The implementations handle the actual encoding and decoding of Messages, so as long as the same implementation is used on the client and the server, game logic on both sides doesn't need to know anything about how the bits are formatted to go over the network. Developers are free to use the provided implementations or write their own. For further server-side abstractions, see the following section's discussion of the MessageManager.

  In addition to Messages between Players, server-side logic components need to communicate as well, perhaps requiring a synchronous response. For instance, assume a Region that is configured to recognize certain areas of the world as trigger-points for game-specific events. When a Player triggers one of these areas the Region doesn't know what it means, only that it has special meaning to the game, so the Region sends an Event to its parent GameMode and waits to hear a response. Another example of this is collision. Some games allow collision, others don't, and some simply don't care. If a Region is implemented to recognize when a collision occurs, it would send a message to the GameMode when this happens and see whether the GameMode allowed this event or not.

  Essentially, Messages are ways to communicate when Players are involved. They are one-way, and need to be encodable to send over the network. Events are for communication between server-side components, demand a response, but don't need any kind of encoding scheme.

- **Item**

  One final interface in Slipstream is Item. This isn't as well-defined yet, but it's pretty clear that most games have a notion of Items and Player inventory. To this end, Item is really just a place-holder for any passive entity that needs to be used on the server. Like Players, Items may join or Leave a Region. Unlike Players, Items will typically not be proactive or take actions of their own.

  Item is defined mostly to help with building re-usable Region, GameMode and Player implementations. These three components will all have to deal with Items, though it's not clear that any of them will need to know the details of a given Item or how it is used. Again, feedback here would be very welcome, as would example implementations of Items in a real game.

This part of the guide has covered all the building blocks provided by Slipstream. The next part will go into more detail with specific implementations against these interfaces.

As a developer, it's important to note that these interfaces leave you a few very specific tasks to get a game developed. You will need to implement a Game object as a coordination point, and at least one GameMode describing an element of play. You will need a UserPlayer implementation, possibly an extension of BasicUserPlayer that merely hooks into your Game implementation. If your GameMode has some notion of physical spaces then you will also need a Region implementation. If you have server-side behavior, then some MOBPlayers will also be needed.

This should leave developers with some reasonably clear, separable tasks. From here it's our hope that developing a game that scales well and expands easily should be much easier. Even if you're not using the Slipstream code itself, hopefully the layout of this design will help with your own game projects.

II. *Slipstream: Programming to the Interfaces*

The previous part covered all the core interfaces of Slipstream. These interfaces have the goal of supporting good game design and also enforcing good use of Project Darkstar. On their own, however, they may seem too abstract and may not make it clear how to use them effectively. This part of the guide covers some example uses of Slipstream, ending with a complete (if questionably "fun") game. Hopefully by the end you'll have a better sense both of how to use Slipstream and how the basic rules from the previous section were applied here.

Note that the examples here aren't really intended as code you would use in a real game. They're intentionally very simple, implementing only the most basic features required to show the Slipstream interfaces in use. There are two implementations of re-usable components, and them some game-specific logic that takes advantage of these components.

IIa. *Re-Usable GameMode: A Lobby*

Let's start with a very simple and common game component[9]. A Lobby is just what it sounds like: it's a place where players first join a game, meet up with other players, and move on to more specific game play. Some types of games make heavy use of lobbies (instanced games with small groups, worlds that are made of many different kinds of games, largely chat-based social worlds, etc.) while others tend to start players directly in some part of the wold. For this example, try not focus on the type of game that would use this lobby, but the more general idea that this is a starting point for filtering players into the larger game.

The goal is to implement a GameMode that will be generally useful to any game. This means that as a starting point, we need to implement GameMode.

```
public class LobbyMode implements GameMode, Serializable,
                                  ManagedObject
{
    private final String modeName;

    public String getName() {
        return modeName;
    }

    public GameProxy join(UserPlayer player,
                          LeaveNotificationHandle handle)
    {
        // …
    }
}
```

While a Lobby may need many different features, there are two common ones: the ability to chat with other players and the ability to find and enter a specific game. Recall that there is a Group that makes chat pretty easy to support.

```
private final Group chatGroup;

/** Create an instance of a lobby. */
public LobbyMode(String lobbyName) {
    this.modeName = lobbyName;
    chatGroup = new UnmoderatedGroup(lobbyName + ":ChatGroup", false);
    // …
}

public GameProxy join(UserPlayer player,
                      LeaveNotificationHandle handle)
{
    chatGroup.join(player);
    // …
}
```

9 The examples here are intentionally leaving out some code to simplify the document. The full, usable examples are available online.

To keep track of the available games, LobbyMode defines methods to add or remove entries by name (not shown here). These will be the names of GameMode instances that the Players can join directly. LobbyMode also maintains a second Group used to broadcast updates about these available games.

```
private final Group gamesUpdateGroup;

public LobbyMode(String lobbyName) {
    // …
    gamesUpdateGroup =
        new SingleSenderGroup(lobbyName + ":UpdateGroup", false);
}

public GameProxy join(UserPlayer player,
                      LeaveNotificationHandle handle) {
    // …
    gamesUpdateGroup.join(player);
    sendAvailableGames(player);
}
```

In order for the Player to interact in this GameMode, there needs to be a GameProxy implementation. In this case, the Proxy will need to keep track of the Player, the Lobby, and the handle used to notify the Game when a player is leaving the Lobby.

```
public GameProxy join(UserPlayer player,
                      LeaveNotificationHandle handle)
{
    // …
    return new LobbyProxy(this, player, handle);
}

static class LobbyProxy implements GameProxy, Serializable {
    private final ManagedReference<LobbyMode> lobbyRef;
    private final ManagedReference<? extends UserPlayer>
                                                playerRef;
    private final ObjectWrapper<? extends LeaveNotificationHandle>
                                              wrappedHandle;
    LobbyProxy(LobbyMode lobbyMode, UserPlayer player,
               LeaveNotificationHandle handle)
    {
        lobbyRef = AppContext.getDataManager().
            createReference(lobbyMode);
        playerRef = AppContext.getDataManager().
            createReference(player);
        wrappedHandle =
            new ObjectWrapper<LeaveNotificationHandle>(handle);
    }
    // …
}
```

Note the use of ObjectWrapper here. Because there are many Slipstream interfaces that might optionally implement ManagedObject, this utility is included to hide the detail of whether a given object implements ManagedObject. In practice it makes the code much simpler, with very little actual overhead.

Finally, the Proxy needs to handle any Messages sent from the client, including notification that the client logged out. [Note: the loggedOut() method should be removed and handled as just another message, since not all Players can login or logout, so this message doesn't always make sense. This will be updated soon.]

```
static class LobbyProxy implements GameProxy, Serializable {
    // …
    public void handleMessage(Message message) {
        if (message.getMessageId() != ModeChangeMessage.STANDARD_ID) {
            return;
        }
        String modeName = ((ModeChangeMessage) message).getModeName();
        wrappedHandle.get().leave(modeName);
        lobbyRef.get().leave(playerRef.get());
    }

    public void loggedOut() {
        lobbyRef.get().leave(playerRef.get());
        wrappedHandle.get().leave(false);
    }

    // …
}
```

The leave() method on LobbyMode is private, and simply removes the Player from the char and update groups. This is a common pattern, where the Proxy will call private methods on its GameMode when its state needs to change.

This shows a complete GameMode implementation. The LobbyMode class can now be re-used for any Slipstream game that needs to support a basic lobby as an entry point. Of course a single lobby will never scale to large numbers of players, but it's a great way to get a game started.

IIb. *Re-Usable Region: OneDimensionalRegion*

Another component that most games will need is at least one Region implementation. Again, the goal of the Region interface is that a Region can be implemented completely separately from a GameMode, so that a developer can share solid Region code with anyone trying to build a game. The example here includes a complete a Region that is by design simple enough that the code is pretty small but unlikely to be useful in many real games.

OneDimensionalRegion is pretty much what it sounds like. It supports all the rules of interacting on a line. The line has some fixed length, and walking off either end of the line results in a notification that the Player has left the Region. A Player may move left or right on the line, and if there are any possible collisions the parent GameMode is asked about what to do. While this is

a pretty limited space, it supports a surprisingly interesting set of possible game play mechanics (as discussed in the next part). More to the point, it illustrates a fairly complete implementation of Region.

As with the LobbyMode example, the complete code is not included here. What follows are the key elements for implementing Region. The complete implementation is available online along with the other examples.

To make this example as simple as possible, we'll define a Region with a single Group that is updated whenever someone moves, joins or leaves. In other words, the scope of what a Player can see is the entire line.

```
public class OneDimensionalRegion implements Region, ManagedObject,
                                             Serializable
{
    final ManagedReference<SingleSenderGroup> groupRef;

    public OneDimensionalRegion(String regionName,
                                long regionLength)
    {
        groupRef = AppContext.getDataManager().
            createReference(new SingleSenderGroup(regionName +
                                                  ":updateGroup", true));
        // …
    }

    public RegionProxy join(Player player, Coordinate location) {
        // …
        groupRef.get().join(player);
    }
}
```

To support the geometry of the world (such as it is) there is a class called Line. Line is essentially a Linked List of points that are currently occupied by Players. It provides a method to add a Player to the Line which returns a LineEntry, similar to the proxy pattern used elsewhere. Movement, finding neighbors and Line leaving is done on LineEntry. Most of the details are skipped over here except for one.

When movement is requested, the Line needs to see if the move is valid. Lateral movement is enforced by the Region, but what happens on collision is an aspect of the game logic. If one Player is trying to move onto a point where another Player is, the GameMode needs to be asked if this is valid:

```
boolean allowCollision(Player player, Coordinate point) {
    CollisionEvent event = new CollisionEvent(point);
    return player.getGameProxy().handleEvent(event).eventAccepted();
}
```

What's important here is not so much the collision handling (since many games won't care about collisions) but the how the event handling is delegated. The fact that a collision occurred

is something the Region knows. How this is handled, however, and whether the movement is allowed (e.g., one of the players dying, or losing hit points, or teaming up, etc.) is inherently a game-specific detail. When you implement Regions in Slipstream, you need to make this distinction clear in order to make your Regions as usefully re-usable as possible.

Given a Line class, the Region code is updated to use this representation of the space:

```
public class OneDimensionalRegion /* … */

    public OneDimensionalRegion(String regionName,
                                long regionLength)
    {
        // …
        lineRef = AppContext.getDataManager().
            createReference(new Line(length));
    }

    public RegionProxy join(Player player, Coordinate location) {
        LineEntry entry = lineRef.get().add(player, (long) location.x);
        if (entry == null) {
            return null;
        }
        // …
    }
}
```

It's possible that the Player won't be allowed to join the Region, in which case the GameMode can decide whether there's somewhere else to the put the Player, or whether they're just not allowed to join the Mode. In practice, it's good design to at least let the player get into a game. For instance, even if the rules of your Game are that a given space on the line can only be occupied by one Player, you might say that the starting point can be occupied by any number of players.

The next thing to implement is a RegionProxy. In this case, the proxy will keep a reference to the Player and Region, as well as the current LineEntry. This will give the Proxy the minimal detail that it's likely to need for almost any Message, and make sure that it can handle most Player-related requests locally.

```
public RegionProxy join(Player player, Coordinate location) {
    // …
    return new OneDRegionProxy(this, player, entry);
}

private static class OneDRegionProxy implements RegionProxy,
                                                 Serializable
{
    private final ManagedReference<OneDimensionalRegion>
                                       regionRef;
    private final ManagedReference<Player> playerRef;
    private ManagedReference<? extends LineEntry>
                                         currentPointRef;
```

```
        // …
        OneDRegionProxy(OneDimensionalRegion region, Player player,
                        LineEntry startingPoint)
        {
            DataManager dataManager = AppContext.getDataManager();
            regionRef = dataManager.createReference(region);
            playerRef = dataManager.createReference(player);
            currentPointRef = dataManager.createReference(startingPoint);
        }

        // …
    }
```

The RegionProxy interface has several methods supporting movement, queries, and messages. The full code shows a complete implementation, but for simplicity here are two of the (possibly) more interesting methods: move(Coordinate) and release().

The move method is used to move the Player associated with the RegionProxy to some new Coordinate. In this one-dimensional region only movement along the x-axis is allowed, so the y and z values are ignored. This implementation only allows moving one point at a time, but in general this method could be used to move greater spaces at once. This implementation enforces some basic rules about movement, then asks the Line to try the move, and reacts accordingly. Among other things, it's the Proxy that handles the possible case that the Player has now left the Line, since this will require notification to the Region.

```
    public boolean move(Coordinate location) {
        if (Math.abs(currentLocation - location.x) != 1) {
            return false;
        }

        LineEntry newEntry = null;
        if (location.x < currentLocation) {
            newEntry = currentPointRef.get().moveLeft(playerRef);
        } else {
            newEntry = currentPointRef.get().moveRight(playerRef);
        }

        if (newEntry != null) {
            if (newEntry.position == currentLocation) {
                // we didn't actually move
                return false;
            }
            currentPointRef = AppContext.getDataManager().
                createReference(newEntry);
            // See MessageManager description for a cleaner approach
            sendToVisible(new MovementMessageImpl(currentLocation,
                                                  playerRef.get()));
        } else {
            // this means that they left the region
            playerRef.get().getGameProxy().
                handleEvent(new LeftRegionEvent(location));
```

```
        }

        return true;
    }
```

Finally, the release() method is called on the proxy if the Player leaves the Region. This could happen because the UserPlayer logged out, the Player died, etc.

```
    public void release() {
        currentPointRef.get().remove(playerRef);
        // as in the lobby, this could be implemented as a leave() method
        // on the Region, but the code is embedded here for simplicity
        regionRef.get().groupRef.get().leave(playerRef.get());
    }
```

Again, the full code is available for this Region implementation, but the remaining methods should be pretty straight-forward. Clearly, one of the goals of Slipstream is to define some common interfaces for others to implement against. The Region and GameMode implementations here are provided partly as examples of how to use the APIs, but also as framework for the game example that follows. Ideally, developers in the Darkstar community will pick up the Slipstream APIs and provide re-usable implementations that make it much easier and faster for developers to build games for Project Darkstar.

IIc. *Game: LineWorld*

Given the LobbyMode and OneDimensionalRegion implementations, a developer could build some games pretty quickly. What follows here is a complete, if somewhat unexciting game called LineWorld. When a client logs in they find themselves in the Lobby. From here they can pick a game, which consists of a single line that they can walk across. While on that line they can chat with others on the line, and inhibit other's progress. When they walk off either end of the line, they go back to the Lobby.

No, not very exciting. The game is kept simple to illustrate the main points of the APIs. It's also a reasonable starting point, however, for thinking about richer functionality that Slipstream makes easy to add. For instance, what about a LineMonster that moves randomly? Or the ability to teleport randomly or "jump" another player some limited number of times. An Item could be employed to let a player move through another player. Hit points could be deducted on collisions. Lines could be connected in a random pattern to create a maze of sorts with items to collect. While the basic game is simple, it should provide entertaining and educational to extend.

To start, a second kind of GameMode is needed to support play on a line. By now, this design should look pretty familiar. There is a Region to support the line, and a Group to support chat on the line. There's also a definition of where all Players start on the line (at the left end-point).

```
    public class LineMode implements GameMode, Serializable,
                                     ManagedObject
    {
        private final ManagedReference<? extends Group> chatGroupRef;
        private final ManagedReference<? extends Region> regionRef;
```

```
    private final static Coordinate START =
        new Coordinate(0, 0, 0);
    // …

    public LineMode(String name, int length) {
        // …
    }

    public GameProxy join(UserPlayer player,
                          LeaveNotificationHandle handle)
    {
        chatGroupRef.get().join(player);
        return new UserLineModeProxy(player, regionRef.get(),
                                     this, handle);
    }

    private void notifyLeft(UserPlayer player) {
        chatGroupRef.get().leave(player);
    }

    // …
}
```

The proxy for the mode should also look somewhat familiar. The difference between this proxy and the Lobby implementation is that now there's a Region to keep track of.

```
static class UserLineModeProxy implements GameProxy,
                                          Serializable
{
    private final ManagedReference<? extends UserPlayer> playerRef;
    private final ObjectWrapper<? extends RegionProxy>
                                          wrappedRegionProxy;
    private final ManagedReference<LineMode> modeRef;
    private final ObjectWrapper<? extends LeaveNotificationHandle>
                                          wrappedHandle;

    /**  */
    UserLineModeProxy(UserPlayer player, Region region, LineMode mode,
                      LeaveNotificationHandle handle)
    {
        DataManager dm = AppContext.getDataManager();
        playerRef = dm.createReference(player);
        wrappedRegionProxy =
            new ObjectWrapper<RegionProxy>(region.join(player, START));
        modeRef = dm.createReference(mode);
        wrappedHandle =
            new ObjectWrapper<LeaveNotificationHandle>(handle);
    }

    // …
}
```

For a LineMode there is only kind of message that we support: movement. If the client sends a move message, then this is handled by the Region.

```
public void handleMessage(Message message) {
    if (message.getMessageId() != MovementMessage.STANDARD_ID) {
        // TODO: this might send a rejection message to the client
    }
    MovementMessage mm = (MovementMessage) message;
    if (! regionProxyRef.get().move(mm.getLocation(),
                                    mm.getSpeed()))
    {
        // TODO: this might send a rejection message to the client
    }
}
```

In the current APIs, there are two kinds of Events that could be raised due to a move: collision with a player or leaving the Region. Collision is not allowed in LineWorld, but any other event is ok.

```
public EventResponse handleEvent(Event event) {
    switch (event.getEventId()) {
        case CollisionEvent.ID:
            return BasicResponse.DENY;
        case LeftRegionEvent.ID:
            handleLeft(true);
    }
    return BasicResponse.ACCEPT;
}
```

The handleLeft(boolean) method called above is a utility shown below. It's called out because the Player may leave the line by walking off an end or by logging out. The mode needs to know which case happened, because in the latter case, the mode doesn't want to tell the Game to put the player back into the lobby.

```
public void loggedOut() {
    handleLeft(false);
}

private void handleLeft(boolean stillPlaying) {
    regionProxyRef.get().release();
    modeRef.get().notifyLeft(playerRef.get());
    wrappedHandle.get().leave(stillPlaying);
}
```

That's the complete LineMode implementation. To finish the game, there are two pieces still needed: the UserPlayer and the Game.

Each Slipstream game will define its own UserPlayer. Partly this is because each game's notion of what a player has, can do, etc. will be different. It's also because the current mode needs to be coordinated by the Game and the UserPlayer. Because LineWorld doesn't have any custom requirements for players, the utility class mentioned before takes care of almost all the

functionality. Here is the private implementation used by the Game.

```
static class LWUserPlayer extends BasicUserPlayer
    implements Serializable
{
    LWUserPlayer(String name) {
        super(name);
    }

    void changeMode(GameMode newMode) {
        setGameProxy(newMode.
                     join(this,new LWLeaveNotificationHandle(this)));
    }
}
```

Finally, there is the Game implementation. Every Slipstream game implements the Game interface, and identifies the implementation to Slipstream via a Java property[10]. Here, the Game will keep track of the Lobby and some number of lines that Players can join. A real game would probably be driven by an external configuration, but to keep this example small some values are hard-coded.

```
public class LineWorld implements Game, Serializable {
    private static final String LOBBY_NAME = "lineworld:mode:lobby";
    private static final String LINES_NS = "lineworld:mode:lines:";

    public LineWorld(Properties p, RegionFactory factory) {
        LobbyMode lobby = new LobbyMode("lobby");
        DataManager dm = AppContext.getDataManager();
        for (int i = 0; i < 10; i++) {
            String name = "line" + i;
            dm.setBinding(LINES_NS + name,
                          new LineMode(name, 400, 5, factory));
            lobby.updateAvailableMode(name, 0);
        }
        dm.setBinding(LOBBY_NAME, lobby);
    }

    // …
}
```

The Game is responsible for providing UserPlayer instances, and providing a way for these players to join the game. Using a utility from the Slipstream codebase, this is fairly simple. Note that in a larger game you should think about having multiple entry points, and in a more persistent world you might pick the starting point based on where the player was last active.

```
public UserPlayer getUserPlayer(String name) {
    UserPlayer player = NameMappingUtil.getUserPlayer(name);
    if (player == null) {
        player = new LWUserPlayer(name);
        NameMappingUtil.addUserPlayer(player);
```

10 GAME.PROPERTY.NAME [note this a placeholder until more general property names can be decided.]

```
        }
        return player;
    }

    public GameProxy join(UserPlayer player) {
        return ((LobbyMode) (AppContext.getDataManager().
                             getBinding(LOBBY_NAME))).
            join(player,
                 new LWLeaveNotificationHandle((LWUserPlayer) player));
    }
```

The only missing piece now is the notification handle used by LWUserPlayer and the join method above. This is the handle that all GameModes use to communicate back to the Game when a player leaves the mode and needs to get into another part of the game. This is similar to the proxies in that it's Player-specific, fairly small, and should cause little to no conflict. Depending on your implementation, this might not need to be a `ManagedObject`, but in this case it is just as a safety precaution (to ensure that the handle is only ever invoked once). This is ok, since it will be invoked rarely, so the extra data store access should have pretty minimal performance impact.

```
    static class LWLeaveNotificationHandle
        implements LeaveNotificationHandle, ManagedObject, Serializable
    {
        private final ManagedReference<LWUserPlayer> playerRef;

        LWLeaveNotificationHandle(LWUserPlayer player) {
            playerRef = AppContext.getDataManager().
                createReference(player);
        }

        public void leave(boolean joinNewMode) {
            AppContext.getDataManager().removeObject(this);
            if (joinNewMode) {
                LobbyMode mode = (LobbyMode) (AppContext.getDataManager().
                                              getBinding(LOBBY_NAME));
                playerRef.get().changeMode(mode);
            }
        }

        public void leave(String newModeName) {
            AppContext.getDataManager().removeObject(this);
            LineMode mode = (LineMode) (AppContext.getDataManager().
                                        getBinding(LINES_NS +
    newModeName));
            playerRef.get().changeMode(mode);
        }
    }
```

Note that in the first leave method, because of how the Game works, this can only be a Player leaving a line and therefore returning to the Lobby. Likewise, the second method must be a request to join a specific line.

This section has covered the Slipstream APIs, some background on how they were designed, and some examples of how to use them. The LineWorld code above represents a complete game built against these APIs. Notice that there's very little direct reliance on the Darkstar APIs, and very few places where the issues from the previous section need to be considered by a game developer, though obviously there are still some potential sticking points. Hopefully this section has been a useful illustration of how to apply some of the basic best practices. The next section will extend the Slipstream examples from this section, showing how custom Services can be used to create even more flexible applicaitons.

3. Extending Slipstream with custom Services

The previous section introduced the Slipstream code, and showed some examples of how it can be used. This section talks about two specific ways of extending Slipstream functionality with custom `Services`. The first is used for message-handling, and the second suggests a way of scripting MOB behavior.

A `Service`, at its most basic, is a node-local element in Project Darkstar that can work inside and outside the transaction model, and can choose to export some or all of its transactional functionality to Application code with a `Manager`. `Services` can see most of what happens on a given node, and the state of the other nodes in the system. It has a richer but much more complex API and contact than Application code has access to. As such, developers should not generally try to write code at the Service level.

This said, sometimes using a custom `Service` is a good design decision. In the case of both `Services` described in this section, state is initialized by non-persistent data, can be processed locally and is not affected by the objects in the data store. Neither `Service` needs a particularly long time to operate, though that's another common reason for using `Services`.

The complete code for both Services is provided online. What follows here is more a discussion of the design, how the Services are implemented, why they make sense as these separate components and how Slipstream can take advantage of them.

I. *Message Service*

In Slipstream all communication between Players, clients and server logic is done with Message instances. Aside from making it easy to communicate between server-only behavior and connected clients, this abstraction means that the details of how Messages are encoded and decoded is separate from the application logic. This makes it easy to change this mechanism as needed, makes the design of your application (arguably) easier to follow and update, and makes it possible to add some interesting optimizations.

Recall from the examples in Section 2 that Message instances were created by directly instantiating one of the shared Message implementations. While this is already doing a decent job of separating message encoding from handling the message content in game logic, it ties

the application to a specific implementation. It does have the nice property of having very little overhead, and in practice is quite fast. The goal of the MessageService and MessageManager is to stay close to this efficient, but decouple this direct dependency. The use of a `Service` to accomplish this is key for a few reasons.

Generally, applications do not need to keep encoding rules in the data store. Instead, the logic tends to be static and can be kept separate from the persistent state of the application. All the application needs is a way to make sure that the message encoding details are loaded on each node, and the `Service` interface provides this initialization step.

One good thing about implementing this behavior at the Service level is that it doesn't require storing immutable state in the data store, and therefore paying the penalty of fetching this data each time a message needs to be handled. It also makes it easier to evolve message handling, or define different nodes in the systems that, due to the implementation of the network, might need to handle messages differently. It also makes it possible to test different encoding schemes (e.g. for performance or flexibility) without modifying any application code or having to update state in the data store.

The model taken with the Slipstream `Service` has three basic elements: registration, decoding and encoding. Ideally application code should never have to deal with any of these directly. The MessageService can still take advantage of the standard Message implementation classes, but this becomes a configuration detail.

On startup, the MessageService reads a file that maps message identifiers to implementations of a new MessageHandler interface. All Slipstream messages sent between the server and clients start with two bytes (a short) identifying the message's type, so when a Message arrives at the server, the first thing the MessageService can do is figure out the message's type identifier, and use that to invoke the appropriate MessageHandler.

For the standard Messages, typically all that a MessageHandler will do is delegate to the appropriate Message implementation. This can be done by using the MessageHandler implementations provided in the server codebase. Beyond this, developers are free to do any additional handling they like, but note that this simple delegation scheme means that there's virtually no overhead in handling this message. The new Message instance is returned directly to the caller and can now be interpreted in game logic.

A MessageManager interface provides access to the MessageService. It allows application code to invoke this behavior. For instance, all messages that arrive from clients are processed by the MessageService (in the UserListener class) so that a Message instance can be created and passed to the client's current GameProxy for processing. This looks like:

```
playerRef.get().getGameProxy().
    handleMessage(AppContext.getManager(MessageManager.class).
                  decodeMessage(message));
```

As you can see, there's nothing specific to the Message implementation in this logic, and it's straightforward for the application code to handle the incoming message. From the point where the GameProxy gets called, all it deals with is the specific Message interface.

On the other end of this process, when server code needs to send a Message to a Player or Players, the MessageService provides a method for creating this Message. An instance of MessageSpec is used which contains the details that are needed to create a specific kind of Message. There are utility implementations of MessageSpec for the standard Message types, and these can be instantiated directly because they are just contains specifying what the Message will communicate.

The MessageSpec is given the MessageService, which once again looks up the appropriate MessageHandler and asks it to create a Message instance based on the provided specification. This has a little more overhead than the decoding operation, but is still fairly inexpensive (especially measured against the cost of everything else happening in the system). It results in an instance of Message that can now be provided to any Player.

```
MessageSpec spec = new ChatMessageSpec("here is a chat message");
Message message =
    AppContext.getManager(MessageManager.class).createMessage(spec);
```

Recall that when this Message is “sent” to a Player, this doesn't always mean that it goes over the network. It may be getting passed to one or more MOBPlayers, for instance, and so keeping the Message in an object-form makes a lot of sense. Note also that this Message might need to be sent to multiple clients, and they may not all be on the same `Channel` meaning that the Message might need to be encoded multiple times. The common implementations provide some basic caching to address this, making it faster to handle multiple sends while still being able to hold this Message across transactions.

Note that because of the way the internals of both the sessions and messages are hidden, it should be possible to extend this model for some further specific optimizations like delaying message encoding until it looks like a transaction might commit, or caching commonly-sent message or message-fragments for re-use. This hasn't been implemented yet, but these changes can be tried without affecting a Slipstream application.

Currently Slipstream implements only some basic Messages and handling or caching routines, but as the next section suggests, this would be a an excellent place to contribute to this project. Even if you're not interested in contributing or using this code directly, hopefully this design will prove useful in your applications. If you do take this design approach, don't forget that each Service will be node-local, so it's imperative that your Services are setup the same way with the same, immutable state or your application will behave unpredictably.

II. *MOBScriptService*

Shifting topics, a common request for Darkstar is a way to write scripts that drive server behavior. This has a number of complexities, mainly focused on the transactional nature of Darkstar, the use of a global data store and the time-limits on tasks. Certainly trying to fetch and compile/run scripts as needed has some significant overhead, and many scripting languages have their own notions of how asynchronous or continuation programming works.

A bigger question here is why and under which circumstances you would need or want to write

scripts as opposed to extending the code directly. It would be nice to allow anyone who doesn't understand Java to write complex logic on the server, but it would be extremely hard to provide some complete system that didn't result in significant performance problems. Likewise, game designers (people who design game logic but are not programmers) often want to be able to add or modify behavior. Again, if the designer doesn't have some understanding of the system, it would be easy to introduce non-scalable solutions in a wide open environment.

One compelling use-case that starts with a fairly small goal is being able to add short, simple, reactive behaviors to MOBs. That is, when a MOB decides to move, or receives a Message, have a static script that react in some limited set of ways. This would be useful for quick prototyping in different languages, and for designers who have limited programming ability. It might be generally useful as well as a way to define and evolve some basic behaviors that are decoupled from the application code, making it easier to use external tools to manage these scripts.

Using Slipstream as a starting point, the Service described in this part was built thinking about this very specific kind of functionality. The goal was to allow scripts that could handle specific Events or Messages for MOBs. All the usual rules apply: the script is executed in a transaction, it cannot have side-effects if the transaction fails and it cannot run longer than the transaction allows. The script won't have access to the general Project Darkstar API, and can't maintain its own dynamic and persistent state, but it can make some specific calls exported by the `Service`. In theory this could be useful for arbitrary design, but as a simpler starting point, we'll assume MOB behavior.

To support this, a new `Service` is created called the MOBScriptService. It pre-initializes a JSR233 scripting engine (the plugin mechanism provided through the javax.script interface) supporting an arbitrary scripting language, and uses this to pre-compile a set of scripts that will be available for MOB behavior. Because this is trying to support arbitrary languages, it does not take advantage of the `Callable` interface (the ability to call directly into specific methods of a script), nor does it allow for arbitrary calls back into the server or access to objects of arbitrary type. Pre-compilation is a pretty strong requirement to make the scripts run quickly[11], so the one requirement of the scripting engine is that it support compiling its scripts.

Because all engines and languages do not support the `Callable` feature, all scripts are assumed to be flat. That is, they don't define any specific interface or set of methods. Instead, when they're called there are some environment variables setup by the MOBScriptService but the script is taken as a single routine starting at the beginning of the file.

When the MOBScriptService starts it looks in a root directory for all files containing scripts for the given language. It tries to load all of these scripts, noting any that fail to compile. The name of the file is also used as the identifier for the script, allowing future calls to reference the named behavior. Once all the scripts have been loaded, the MOBScriptService is ready to process requests from the MOBScriptManager.

---

11 Some basic observation is that even a very minimal javascript, ruby or lua script can easily take on the order of 20 milliseconds to compile. Running the same script once compiled, however, will usually take less than 1 millisecond. This will vary based on the engine, but the general observation is that compilation is a necessity.

```
// assume this is part of the implementation of a MOBPlayer
private static final String MOB_NAME = "FriendlyVillager";
// …
AppContext.getManager(MOBScriptManager.class).
    callScript(MOB_NAME, this);
```

One of the pragmatic challenges of using this approach is that different engines have different rules about how multiple threads and reentrancy is handled. In practice, this may mean doing some strange startup work in your Service to make sure that multiple scripts can be run simultaneously. You can see this in the MOBScriptService implementation. A fairly simple startup routine worked for the default Rhino javascript engine. To support a pure-Java lua engine that isn't multi-threaded, however, separate instances of the engine were required for each thread that might invoke a script, and each of these engines needed to pre-compile its own copy of the script. Other engines (e.g. python) seem to exhibit the same behavior. For a small number of scripts this isn't too problematic, but could cause significant scaling problems for large numbers of behaviors.

The current MOBScriptService implementation uses the example of some static methods implemented directly on the class to provide ways for the script to call back into the server. While there are other, engine-specific mechanisms for supporting this kind of feature, in practice this is probably the most broadly re-usable approach and helps to define some narrow functionality to support scripted MOB behavior. The MOBScriptService keeps track of the MOBPlayer running in the current thread as a way of accessing the MOB state when any of these static methods is called, which assumes that the calling thread will be “owned” by the calling MOBPlayer. This is part of the reason that this `Service` is targeted at supporting behavior of a single calling MOBPlayer.

Note that the current implementation is little more than a proof of concept that something like this can be built within the Project Darkstar model, and that it will run within the required boundaries. The current MOBScriptService would need to be extended to be useful to any but the most basic of applications. This said, it should prove useful as an example and starting point for anyone interested in this kind of scripting approach.

There are other scenarios where Services will prove useful, but hopefully the MessageService and MOBScriptService examples are helpful both in writing similar functionality and understanding when you need to work at this level and what this requires. If at all possible, it's best to avoid Services and keep your logic at the Application level. If you do need to write a Service, try posting to the forums, where there are lots of folks with experience to help you out. In the meantime, these Slipstream Services should act as useful examples, and should be re-usable in applications that you are trying to write.

4. Open Issues and Next Steps

This section describes most of the missing pieces and possible next steps for Slipstream. Since Slipstream was started as an experiment to apply what we thought were some best-practices and useful patterns, the initial goal was not to build a full implementation. The current codebase

provides some strong examples of how to build a scalable application for Project Darkstar, and can be used today to start prototyping a new game.

This said, a lot of work would be needed to turn Slipstream into a full project that is generally useful for real applications or shared components. The hope is that there is enough of a starting point that other developers in the community will be interested in contributing, and will be able to find some specific, separable areas to contribute. There's plenty to do, but what follows are some general starting points.

Client Support

Currently most of the work has been on the server. This means that there are no client libraries to handle messages, send commands, etc. The Message definitions and implementations are part of a shared package, so much of the work to handle Messages on the client is in place. What there isn't, however, is any kind of caching for PlayerID (see the next item), any factory for message creation like the MessageManager on the server, or any utilities for reacting to common events.

More importantly, there is the Entity interface which tells a client who sent them a message, but no automated infrastructure on the client or the server for responding directly to that Entity. This would make client-side coding much easier and more natural, and should be fairly easy to add. Clients would probably need a standard DirectedMessage or something of the sort, and server code would need an interface to call back used to decide how or if these messages are handled.

Essentially, while the building blocks are there to write the client-side of a Slipstream application, there is no natural API like what is provided on the server. Providing this kind of framework would make writing clients much more compelling.

As an obvious extension of this work, currently the Message implementations are only in Java. It would be great to have implementations in other languages of the standard Slipstream messages, and a framework for those languages, so that non-Java clients could easily work with a Slipstream server.

Message Handling and Entity Identification

Basic message handling is currently provided by the stadnard Message interfaces and implementations, and the MessageService on the server. The Service could be extended as described in the previous section for optimization, but the basic framework is solid. Likewise, better encoding techniques would be great to explore, though the basic code provided should be sufficient to help developers get started.

What is less-well defined is how Entities are defined in a generic way between clients and the server. The current design is that all Messages can identify a sender. On the server, this sender is typically a Player, which makes it convenient to interact directly with the Player logic. When a Message is sent to the client, or when a client wants to send a message to a specific Entity, however, an integer is included in the encoding to specify the Entity.

How the client and server know the identifier-to-Player mapping is still being fleshed out. Open questions include whether these are permanent or temporary identifiers, whether UserPlayers can or should be identified differently than MOBPlayers and how often clients will need to ask for details about this mapping. This is clearly a key topic to making the client-side coding easy, so it's an important issue to address. In the current server code there are place-holders for the identifier and ways to pass around sender/Entity details, but this needs more work to enable a full client.

More Message and Event Types

There are currently only a few “standard” types provided in the Slipstream codebase. These are useful, but clearly not enough for getting started on a real game. Some work is needed to decide which Messages or Events are common enough that they should be part of the core, and where additional types should be collected. It would make sense to have a common repository or types for general use, since this would make re-use of different components much easier.

Region Implementations

Perhaps the most important utility missing for Project Darkstar developers right now is flexible, scalable, re-usable Region support. Several common types of Regions are definitely needed to make Slipstream and Project Darkstar really useful. Some examples would be flat mesh-based spaces, three-dimensional space, or even simple grids.

Design for these kinds of Regions has been discussed on the forums, so there is alredy a good set of examples to start this development. Also discussed on the forums is how such an implementation would work with some graphical authoring tools used to define world spaces. There are some community projects to build parts of these utilities, but nothing generally re-usable. Building on the Slipstream interfaces, if there were even some basic Region implementations it would be of great benefit to the community as a whole.

Adding to this, world building tools that work with these Regions would be even better. If one tool could be used to design a world and then export the details needed for the server-side representation, it would be of immense value to the community.

Server-side Scripting

A common feature of world-authoring tools is the ability to script some behavior. If this behavior could be supported on the server though some set of lightweight script engines, it would make it easier for non-developers to work with Darkstar. Obviously, these scripts will be limited by the transaction and datastore model, but as the exercise from the previous section demonstrates, there are some good ways to think
about script integration.

The real issue here is trying to figure out when it's important to write server-side behavior in something other than Java, who will be writing these behaviors, and what kinds of access will be needed to the server state. Working on this kind of support, especially integrating with world-building tools, would be of real value. The MOBScriptService from the previous section shows one simple, early approach for supporting some behavior scripting, but also shows some of the

problems and short-comings of trying to merge scripting with the Project Darkstar model. More experimentation here would be worthwhile.

5. Conclusion

Hopefully this guide has helped you to understand a little bit more about how the Project Darkstar model works, how to build applications that will scale effectively in that model, and how to think about structuring your code for rich games. It's also our hope that you're interested enough to try out Slipstream, and possibly even think about contributing to the development based on the previous list or some of your own ideas.

If you are interested, please check out the project page for Slipstream:

```
https://slipstream.dev.java.net/
```

Please also go to the Project Darkstar forums and post your questions or suggestions. If there is clear interest, we'll create a new discussion and development area, and start tracking some of the issues and enhancements that people think are useful. Most importantly, please let us know what you found helpful or confusing about this document and the project as a whole. This project is still in its early stages of development, so it will be very useful to know how it can improve and progress.

Thank you, and have fun writing your games!