A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

by Henrik Bäarnhielm ⟨redstar@kth.se⟩
and Daniel Sundström ⟨daniel@monkeydancers.com⟩
and Mikael Vejdemo-Johansson ⟨mikael@johanssons.org⟩

April 23, 2014

In March 2013, the battleship Smålånd in the Gothenburg harbor was transformed into a spaceship, the Monitor Celestra of the Battlestar Galactica fleet, for a three day Live Action Roleplaying Game. To support the feeling of total immersion into the game world, we built extensive technological support for the game experience, including a sound system with a programmable control layer written in Haskell.

In this article we will describe this control layer and our experiences building and using it for this game project.

The Monitor Celestra is a game that transformed a World War II era battleship into a space ship, inviting the participants of the game to enter the Battlestar Galactica [1] universe and forge their own path into the future. The game, a Nordic Style Live Action Roleplaying Game (LARP) assigns roles to the participants – as crew, passengers and refugees on board the space ship – and act out the resulting story within a framework of plots, story lines and clues. The medium differs from theatre plays in the lack of a script and from improvised theatre in the lack of an audience – to consume the medium is to participate in it.

Nordic Style LARPs are characterized, among other things, by a strong focus on immersion. The goal is for participants to feel embedded in the game world without the distraction of the real world disturbing the fiction. In recent years, more and more games have been organized that include technological support to improve the feeling of immersion into the game world.
For The Monitor Celestra, full immersion was implemented using a number of different techniques: the signage of the ship was replaced with signs that were in the Battlestar Galactica aesthetic, the ship had custom-built control panels controlled a space dynamics simulator, and the game master team had the ability to trigger a large array of scripted, partially scripted or improvised events that launch sounds and influence the ship control panels to convey the story.

An important part of this immersion was 15 pairs of loudspeakers, each pair hooked up to a Raspberry Pi [2] and connected to a wired ethernet network. These loudspeakers played localized sound effects as well as an ambient sound backdrop through the entire game, drowning out the sounds of inner city Gothenburg and supporting the game experience by confirming consequences of player actions through mediated responses.

**Background**

LARP is a form of game where participants receive roles and then proceed to enact their roles within a framework of plots, story lines and clues. Traditionally, LARPs have been focused on telling stories in a fantasy/medieval setting, but the form has seen a wider spread of genres over years. LARP has developed into various design schools and style – mostly based on geographical distribution. Thus, there are Nordic LARPs, American LARPs and Russian LARPs, to name a few [3]. Each school of design has its own theories on what constitutes a good LARP and what goals one must strive to achieve with the game design, participation and outcome of the game.

In the same tradition as table-top role playing games and improvisation theatre, LARP is played out by players in real time in the same general physical area. Players wear costumes and make props to better simulate their respective characters. In general, participants receive instructions beforehand regarding specific game rules, other characters, shared background information, etc. Organizers define social mechanics and hard game rules for resolving combat, conflicts or sex.

**Nordic LARP**

The Monitor Celestra belonged to the Nordic LARP school of design. This school could be argued to emphasize a few different points as being major when creating or participating in the experience:

▶ The game should represent the world as faithfully and completely as possible (games which fulfill this criterion are commonly referred to as 360° games.) Everything players see and/or experience represents itself within the game – a gun is a gun, a knife is a knife.
Henrik Bäärnhielm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

- Participants should put emotional engagement and experience above achieving in-game goals. Powerful emotional impacts should take precedence when deciding how to move the game forward.
- The players should control the outcome of the game. While parts of the game might be scripted, the main outcome of the game should be left to the players as far as possible.

Due to the nature of the Nordic school of design, games designed within that school tend to focus on moral choices and complex social interactions rather than traditional fantasy stories. There have been games depicting everything from Danish hobos on the road (The White Road [4]) and students investigating possessed ladies (Prosopopeia [5, 6]), to 1950s families hiding from nuclear war (Ground Zero [7]) and terminal cancer patients (Luminescence [7]).

LARP vs. Similar activities

What separates LARP from other similar activities such as tabletop roleplaying games, computer games and improvised theatre is still an open question. Researchers [8, 9, 10] have pointed to a few main points:

- Emotional engagements increased radically when players are physically engaged in the activity [9].
- Moral choices and consequences from player actions have larger impact when acted out by other physical players [9].
- Players allow themselves more freedom when acting within a ludic circle,¹ thus allowing for a wider range of emotions, experiences and shared actions [8, 11].
- Reflections on the complexity of social and/or humanistic behavior or models yield a higher level of understanding [10]. Research has shown that when trying to understand complex models of human behaviour, both in extreme situations such as emergencies and in everyday contexts - LARP tend to work very well as tool for exploring motivations, social roles and social systems based on incitement.

It has been humorously suggested that LARPing may be seen as the “extreme” sport of interactive experiences due to the efforts involved with staging and participating in games, the impact on participants and the cumulative effects of player actions and organizer design.

¹*ludic circle*: a shared space of playfulness, from *ludic*: playful
Similar Events

There have been other efforts to use technology to enhance LARP experiences within the game as a key part of staging the experience, mainly within the Nordic school of LARP design. Historically, these experiments have been conducted by a small number of game designers and writers. Experiences where technology were used in a similar capacity as on The Monitor Celestra includes:

- **Prosopopeia: Bardo 1** A technology-enhanced LARP played in the autumn of 2005. Players used technology concealed in an old reel-recorder to communicate with dead spirits.

- **Prosopopeia: Momentum** An continuation to the Prosopopeia project, Momentum played in the autumn of 2006. Players used a wide variety of technology, including hardware built using custom constructed components to explore and wage battles against forces on the other side of death.

- **Felsäkert Läge** A technology-enhanced LARP in the autumn of 2010 situated in a future, post-apocalyptic world, staged in an abandoned mining town in the north of Sweden. Technology was used to simulated surviving technology from before the catastrophe, telling stories and providing quests, items and treasures for the players to find and use in their own game.

An overview of similar experiments (and others) can by found in [7].

Immersive Soundscapes

Earlier efforts on immersive soundscapes include various implementations in Python, support in most holistic game engines and similar. However, the main difference between these implementations and the system built for The Monitor Celestra is the spatial scale. Usually, these soundscapes are meant to be enjoyed by a single person in headphones or a handful of persons in a single or a few rooms.

The difference in scale between these implementations and the one described in this paper makes comparisons hard to do. Most issues met by the usual implementations were either deemed irrelevant (such as super-focused spatial placement in the scene – a rough idea was more than enough for a battleship) or too specific (solutions for physical placing loudspeakers in a square room).

---

2 As apart from using technology for support functions, such as economy, participant management etc.
Every LARP differs in structure and design according to the design goals, creators and aims with the project. The Monitor Celestra had three runs with 150 participants each, split over three weekends. The runs were separated from each other and played out the same story with different participants (even though some participants bought tickets for all the runs).

Every run was split into four game sessions of 4–6 hours, with breaks between for sleep and briefings. Food was distributed and eaten while in play. Even though the battleship Småland was equipped with bunks for its crew, fire regulations prohibited sleeping on board the ship. Hence, the participants slept at a nearby hostel between game days.

The story design started out in the setting of Battlestar Galactica: a fleet of refugee spaceships carrying most of the remainder of humanity flee from robot aggressors. In the game, The Celestra is separated from the remaining fleet by accident, and the players have to decide on their actions, while facing various threats: robots hunting them down, colonies with radical political views and demands, and internal conflicts on board the ship. Each of the games ended in a different way, ranging from a valiant effort to save the human race by nuclear suicide taking out a threat before it could reach the rest of the fleet to setting course for deep space and forging a new path for the isolated ship.

Game control system design

The technology team for The Monitor Celestra settled at a relatively early stage on an overall design for the game support technology; the choices made included:

- Wired ethernet communication for everything, split into three different parallel networks: sound, game system, and player hackable. Due to the nature
of the play area, parallel network backbones were installed for maximum durability and error management.

- Communication between game components over AMQP from a RabbitMQ running on a local game server on the main network. Game logic, world engine and game rules were written in Ruby.
- Game consoles and GUIs written in HTML5 Canvas rendered on Chromium running in fullscreen.
- Game consoles able to be used interchangeably with everything being designed for plug-and-play.
- No authentication required within the game network. All published messages can be trusted on a system level.
- A single game component responsible for game world, time progression and synchronization; in this case, a rack server running latest stable Ubuntu LTS.
- Connection to external network and Internet through a single point-of-entry, the main game server.

These choices were made based on experiences from earlier projects staged in similar contexts where the main body of the work was done by unpaid volunteers in uncontrolled and rough environments.

Sound system design

The original specifications for the Celestra sound system were as follows:

- Each section/room of the ship should have an independent sound producer, so that each room could play different sounds simultaneously. Each sound producer should be able to mix several sounds.
- It should be possible to trigger sound effects at specific locations, both automatically via the game control system and manually by game masters. For example, a torpedo sound in the torpedo room is played when a torpedo is launched (triggered by a player), and a warning sound is played in every room when the hull is compromised.
- To maintain immersion, the sound effects must have low latency.
- It should be possible to trigger sounds at certain future times. For instance, it should be possible to specify that a torpedo hatch closing sound is played exactly one second after the torpedo launch sound.

The rest of the game control system was designed to run on a relatively small number of servers in a control room. One option for the sound system would be to have a sound server with one sound card for each section, connected to pairs of speakers located in each section. However, it was decided that a better solution would be to have actual computers at each section, with their own sound cards, connected to a central computer via wired Ethernet, which would already existed.
Henrik Bäärnhelm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

in most rooms. This setup would be easier to build in terms of hardware and would be easier to program for. The computers located next to the speakers in each room then had to be inconspicuous and inexpensive, since quite a large number would be needed. This led us to go for the Raspberry Pi (RP) setup.

The choice of RP added constraints. RP has limited RAM and CPU power; hence, we decided to avoid adding dependencies to the game back-ends on every RP. While most of the game control systems were written in Ruby and communicated over AMQP, we decided to implement a special purpose sound daemon in C++, which would run on each RP and communicate with the central sound server directly via sockets. This would enable us to achieve lower latency and not drain resources from the RP.

**Sound daemon**

All required sounds were stored as sound files on every RP, effectively creating a global sound list $S$. The sound daemon should then accept a few simple commands on the socket:

- Play sound $k$ from $S$ at absolute time $t$, as sound with ID $n$.
- Stop sound with ID $n$, at absolute time $t$.
- Change volume to $v$ on sound with ID $n$, at absolute time $t$.

The idea here was that by combining these commands, it would be possible to realise all the required sound system features. Each RP was time synced to the rest of the network using NTP [12], hence the sound commands could have absolute timestamps with high accuracy. To achieve the possibility of playing multiple sounds at a single RP, the sound daemon communicated with PulseAudio [13], creating one stream per sound being played. To achieve low latency and to simplify the implementation, the daemon read in $S$ to RAM upon start-up, rather than reading a file upon each play command. This added the constraint that the total uncompressed size of the sounds had to fit into RAM of the RP.

**Ambient and story sounds**

It turned out that apart from the sound effect type of sounds that the sound daemon was created for, the game required two other types of sounds.

- Ambient sounds, which should be looped in the background in a section. For instance, in the reactor room there should be a constant sound from the reactor.
- Story sounds, which were played at specific events in the game. These were typically very long compared to ordinary sound effects and were triggered explicitly by the game masters.
These types of sounds did not quite fit into the sound daemon. The ambient sounds required looping and needed to be cross faded into themselves to avoid clicks; the story sounds were too large to fit into RAM. Moreover, the low latency requirement for these sounds were much less strict.

We therefore decided to use MPD [14] for these sounds. MPD reads files from disk, hence the sound size would not matter. It also has built-in cross fade and loop support and can output to PulseAudio, so it was compatible with the sound daemon. Each RP was running an MPD daemon, and the sound server communicated with it using a standard MPD client library.

**Sound daemon controller**

On the sound server, a small piece of controller software was running, which listened to AMQP sound commands, transformed them into sound packets for the daemon or to commands to MPD and sent them off to an RP. It was also responsible for transforming node numbers to IP addresses of RPs.

Like most of the game control software, this controller was written in Ruby. The original idea was that this would be the entry point of the sound system. However, it turned out that its simple interface, while in principle sufficient for all sound purposes, was too low-level for creating the sound scenes that the sound designers wanted to build for the game: crossfades, seamless loops and other sound design features would require a higher abstraction than the play/loop/stop/change volume setup for this interface.

Thus, we were led to introduce the intermediate layer in Haskell, described in the following sections. From now on, we will be referring to the sound server described here as the **low-level system**, and to the Haskell layer as the **high-level system**. The Haskell layer works with two separate interface layers; a **low-level interface** facing the low-level system, and a **high-level interface** facing the game master and sound designer teams.

**Requirements on the sound scene DSL**

The work on building a dedicated domain specific language (DSL) for describing sound scenes started at a relatively late stage of the project, when most other platform decisions had already been made. The need for an intermediate layer emerged from the simultaneous need for low-level simple sound execution protocols that could run on a Raspberry Pi platform and for abstract sound descriptions with a low technical usage barrier to enable the game fiction design and sound design crews to interact with the system.
Henrik Bäärnhielm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

This setting produced a number of requirements that we tried to adapt to during the project:

**Simple output** Output from the DSL to the lower level systems needed to have a simple structure, preferably consisting of discrete orders to **PLAY**, **LOOP**, **STOP** or **ADJUST** sounds at sound nodes (both addressed by integers).

**Accessible input** The DSL needed an easy to read and easy to write format that the artistic side of the project could use to specify sound scenes.

**Fast development** The DSL needed to be developed during 4 months of volunteered free time, during a time span that included several major conference deadlines. This limited the amount of programmer manpower available.

**System compatibility** The DSL needed to react to **AMQP** messages, store state in **Redis**, and format its output in a way easy to parse by the receiving low-level and high-level interfaces.

Due to earlier experiences with Haskell, we chose GHC and the Haskell Platform as a host system for the DSL development. An early decision was to use the automatic parser generation in the standardized Haskell **Read** and **Show** classes; later on, we used the **Generics** extension to automatically generate JSON parsers and formatters in order to plug into the existing **AMQP** message passing architecture. This reduced large amounts of DSL design to designing appropriate Haskell types: writing native Haskell code with custom-built data types that represented the domain allowed for large freedom in defining our own semantics while retaining the full power of Haskell packages and programming styles.

**Design of the sound scene DSL**

From the overall design decisions for the entire sound system and game system, some parts of the design of the sound scene DSL subsystem were clear: we had to interact with a **Redis** database and with an **AMQP** communication system, and we needed to send JSON packages according to a fixed format in order to control the lower level system.

On top of this, we had requirements for swift development, ease of use, and minimizing the amount of extra parsers to write. These criteria were central in selecting Haskell as a platform for the tool: out of the platforms that members of the workgroup were familiar with, Haskell was far more capable of quick development and quick automatic generation of parsers and serializers than all the alternatives.

Our system ended up depending on a family of Haskell packages that cover many of the interoperability requirements. In total, we relied on **aeson** [15],
amqp [16], attoparsec [17], base [18], bytestring [19], containers [20], ghc-
prim [18], hedis [21], mtl [22], regex-posix [23], text [24], and unordered-
containers [25], for all our library needs.

From these preconditions, we decided that the best way to construct a DSL
would be to encode all important information in terms of specific Haskell datatypes,
so that Haskell methods for generating parsers and serializers could be used. Ad-
ditionally, we needed a hierarchy of types bridging the gap between the human-
readable game master facing DSL and the machine facing (already defined) JSON
protocol.

The resulting hierarchy of datatypes that we decided on was:

\[
\begin{align*}
\text{SoundCommand} & \rightarrow \text{SoundSpec} \\
\text{DaemonSpec} & \rightarrow \text{FilterSpec} \\
\text{DaemonCommand} & \rightarrow \text{AMQPDaemon}
\end{align*}
\]

Figure 2: Dependency and compilation path hierarchy for the sound system.

where we had separate compilation steps to transform a SoundCommand into a
DaemonSpec, a DaemonSpec into a DaemonCommand and a DaemonCommand into an
AMQPDaemon.

These types all had different roles:

- **SoundCommand** encoded all orders the system expected from game masters and
sound designers. In order to more easily design datatypes, we separated out
the descriptions of a sound scene and of a reaction trigger from the command
type into the subordinate types **SoundSpec** and **FilterSpec**.

- **DaemonSpec** encoded, abstractly, the order types the lower level system accepts.

- **DaemonCommand** encoded in full detail a single order for the lower level system.

- **AMQPDaemon** was a datatype explicitly constructed to serialize through **Aeson** into
a JSON package that the lower level system could parse.

Here, the **SoundCommand** type encoded all orders that the game masters and
sound designers wanted to be able to give to the system, while **DaemonSpec** ab-
stractly encoded all command types the lower level system accepted.
Henrik Bäärnhielm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

All datatypes derived `Read`, `Show` and `Generic`, which allowed us to automate parsing both from a Read-Evaluate-Print-Loop as well as create automatic JSON parsers and encoders from `Aeson`.

**SoundCommand**

Our separation of the sound scene description from the Haskell layer control commands and the automated reactive triggers builded on their separation into different datatypes. The family of datatypes we designed closely mirrored the task division for the system. At the top-most abstraction level, there was a data type `SoundCommand` enumerating the various commands that can be given to the system.

```haskell
data SoundCommand =
    Define Id SoundSpec |
    Commit |
    Restore |
    Diagnostic |
    ReadState |
    ReadSounds |
    Compile SoundSpec |
    ReadOut SoundSpec |
    Execute Id SoundSpec |
    Trigger Id FilterSpec SoundCommand |
    SoundCommand :+: SoundCommand |
    Declare Id SoundCommand |
    Call Id |
    Delete Id |
    Nop
    deriving (Eq, Show, Read, Generic)
```

There were commands for interacting with the database state storage: `Commit` and `Restore`; commands for debugging and analyzing what a particular sound scene description was interpreted to: `Diagnostic, ReadState, ReadOut`; commands for naming and recalling both sound scenes and entire commands: `Define, Declare, Execute, Call, Delete`; and commands for triggering sound scenes either through events by `Trigger` or through direct command by `Execute`. Finally, there was `ReadSounds` that reported available sound scenes up to a top level UI layer, and `:+:` for chaining commands together as well as a `Nop` that could finish an automatically generated chain of commands for programmatic creation of triggers and action chains.

The definition used the two types `SoundSpec` and `FilterSpec` to parametrize its entries. These were given by
data SoundSpec =
  Play File Segment Time Loudness |
  Loop File Segment Time Loudness |
  RadialDecay File Segment Time Loudness |
  SoundSpec :+: SoundSpec |
  Use Id |
  Stop Int Time |
  Fade Int Time Time Loudness |
  StopId Id Time |
  FadeId Id Time Time Loudness |
  NopS
  deriving (Eq, Show, Read, Generic)

and

data FilterSpec =
  FilterSpec :& FilterSpec |
  FilterSpec :| FilterSpec |
  MatchAll [FilterSpec] |
  MatchAny [FilterSpec] |
  MatchEvent String |
  MatchSender String |
  MatchKeyValue String String
  deriving (Eq, Show, Read, Generic, Ord)

The sound scene specification allowed for playing a sound by name or by index, either once or on an infinite loop, and for stopping and changing volume. These were the operations understood by the low level system as well.

In addition to these, the Haskell layer automatically generated packages to smoothly fade between volume settings (for a spatially distributed decaying sound scape) and for saving and recalling sound scenes. After an initial attempt to design the fades, we ended up building in support for remembering the last set volume for a sound, so that fades could be given by a target loudness rather than by start and stop loudness settings.

The inclusion of \texttt{Nop} and \texttt{NopS} helped us automate generation of lists of actions; together with the concatenation constructions given by \texttt{:+:} and \texttt{:+}, there was a full monoidal structure on both these datatypes, enabling easy generation of composite commands from lists of command parameters.

Since the entire system actively listened to the AMQP traffic of the entire game system, it was easy to include a reactive component: using a simple regular expressions-based recognition engine, we were able to write simple rules that would when matched trigger arbitrary pre-constructed \texttt{SoundCommand} actions. The rules
Henrik Bäarnhielm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP were encoded using the type FilterSpec and their corresponding actions were encoded with the Trigger constructor of SoundCommand. All AMQP messages in the game system contained a sender, an event key and some collection of key-value pairs, all of which could be regular expression matched with the rules specified as FilterSpec entities.

**DaemonSpec**

The DaemonSpec type encoded the abstract payload of a single instruction to the low-level system. An element of type DaemonSpec encoded all the descriptive information needed for a low-level system command, without containing transient information required for emitting any particular command package. In particular, there was a serial ID number assigned to low-level command packages to allow later commands to modify a running sound. These ID numbers were not added in the DaemonSpec representation, but rather in the next lower representation.

```haskell
data DaemonSpec =
  DaemonPlay Int Segment Time Loudness |
  DaemonLoop Int Segment Time Loudness |
  DaemonStop Int Time |
  DaemonSet Int Time Loudness
  deriving (Eq, Show)
```

**DaemonCommand**

The DaemonCommand encoded a message that could be sent to the lower-level system. In particular, the command encoded a sequential id-number used for later modifications of looping sounds and set up a datatype for easy parsing for the receiving system.

```haskell
data DaemonCommand = DaemonCommand {
  node :: Int,
  dcid :: Int,
  sound :: Int,
  time :: Int,
  volume_left :: Float,
  volume_right :: Float,
  command :: Int
} deriving (Eq, Show, Generic)
```
The type `AMQPDaemon` really only existed in order to wrap a `DaemonCommand` item for serialization with `Aeson` and transport in an `AMQP` package. The type is defined as:

```haskell
data AMQPDaemon = AMQPDaemon { 
    devent :: String,
    dsender :: String,
    dcmd :: DaemonCommand
} deriving (Eq, Show, Generic)
```

with custom JSON instances created by

```haskell
instance FromJSON AMQPDaemon where 
    parseJSON (Object v) = AMQPDaemon <$> 
        v .: "event" <*> 
        v .: "sender" <*> 
        v .: "data"
parseJSON _ = mzero

instance ToJSON AMQPDaemon where 
    toJSON ad = object ["event" .= devent ad, 
        "sender" .= dsender ad, 
        "data" .= dcmd ad]
```

These were the only parser and encoder instances we wrote ourselves for this project.

**Persistent state**

There was a number of pieces of information the system needed access to, with various levels of persistence. We designed a tiered state type consisting of a serialisable section and a collection of transient state properties, described by

```haskell
data SST = SST { 
    serst :: SerST,
    dbconn :: R.Connection,
    achan :: A.Channel,
    cmdid :: Int 
}
```
Here, we encoded instance-specific connection data for the Redis database in dbconn, instance-specific connection data for the AMQP communication channel in achan, and an instance counter for sequential command ids in cmdid. The rest of the state was stored in the serst (serializable state) field, which was saved to the database in order to persist settings between runs.

The serializable state in turn was given by

```haskell
data SerST = SerST {
  soundscapes :: M.HashMap String SoundSpec,
  commands :: M.HashMap String SoundCommand,
  triggers :: [(Id,(FilterSpec, SoundCommand))],
  loops :: [(Int, (Segment, Int, Loudness))],
  tags :: M.HashMap String [Int]
} deriving (Show, Generic)
```

where we make extensive use of the strict hashmap implementation from the unordered-containers package.

Here, soundscapes saved all named sound scape descriptions; commands saved all named sound system commands; triggers saved trigger definitions and is iterated through whenever a package showed up that the system might react to; loops saved currently playing loops and their most recently known loudness; and tags saved a lookup table from human readable names to command ids.

In addition to these, there was a pair of hardcoded lists defined in the source code itself: playable and loopable. These two lists contained the names used throughout the sound system to refer to all playable sounds, in an order kept synchronized with playlists on both the lower level system daemon and on the MPD instances. From these were also derived two hashmaps playDict and loopDict to enable faster index lookups given the names.

**Sound scape design daemon**

The library described above was then used by a daemon which ran throughout the game on one of the game control servers and reacted dynamically to instructions arriving by AMQP. This daemon stored the state in an IORef and used the callback structures in the amqp package to listen for and react to AMQP messages. The entire logic of the server was encapsulated in this callback and the functions it called.

The callback function parsed out the payload of the received AMQP package and checked whether it matched a regular expression. If so, it parsed the contained command and acted on it; if not, it ran the package through all defined patterns in triggers and ran the associated action for each matching pattern. This linear lookup may have been slower than more complex solutions, but it had the benefit of being easy and reliable to design and was probably fast enough for this application.
In the end, as we will describe later, there were some latency issues with the system as a whole. Our diagnostics of the trigger list handling were inconclusive, but did not seem to generate the observed latencies.

The function \texttt{action} executed a \texttt{SoundCommand} and carried the entire logic of the system. This is where the data type was translated into actual reactions. Most of the implementations were straightforward: read the current state from the \texttt{IORef} variable, extract relevant parts of the state, and then either assemble a package for \texttt{AMQP} or for \texttt{Redis} and send it out, or modify the running state according to the received instructions. By far the most involved of these was the implementation of the \texttt{Execute} command, responsible for sending out low-level instructions. This command constructed \texttt{DaemonSpec} descriptions, assigned sequential command id numbers, packed the result into \texttt{AMQP} packages, and – depending on the exact type of each command – modified the state to remember the details of the sent commands for later recall when constructing fades or stops.

Large swathes of the daemon code were reused to construct a command line interface that generated \texttt{AMQP} packages for controlling the system, allowing for an accessible debugging and programming interface.

\section*{Usage examples}

In this section, we will give a few example programs written in the DSL, to give a flair for the kind of scene description language that emerged from our design choices.

There were different types of sounds specified by game masters and sound designers, based on the planned usage and design requirements of the sound. There were ambient sound loops used to simulate states such as running the ship’s main reactor or the sounds of massive computer banks running in the sensor processing room. There were feedback sounds triggered in response to participant actions, such as loading a torpedo or triggering a sensor sweep. These sounds mainly functioned as both confirmation to the participant that the system has registered the event and to communicate the occurrence of an event to participants in other parts of the ship. Apart from these sounds, the game masters had access to a wide variety of sounds and noises with no predetermined meaning in order to simulate everything from debris hitting the outer hull to onboard fighting, as well as pre-recorded prologues used to segue players into the game at the start of each game session.

Some sounds needed to run through an Attack-Decay-Sustain-Release sequence, composed of multiple sound files and an arbitrarily long sustain-phase. Other sounds needed to trigger with fixed time intervals.

All sounds needed to be played on a set of speakers – but not all sounds were
Henrik Bäärnhielm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP to be played on all speakers when played. For example, sounds of airlock doors opening were played in the section containing the doors and sections immediately adjacent to it.

We built a setup script to handle all the sound setup in a repeatable and restartable way. The script had a `main` consisting of:

```haskell
main = mapM_ (putStrLn . show) $ <soundlist>
```

where the list of sounds was constructed in bits and pieces. We will explain and give examples for each section of the sound list here.

We also defined and used a single value for simultaneously addressing all sound sites (except the engine room sub-woofer)

```haskell
global = [1..11] :: [Segment]
```

as well as a utility command to help string together sound declarations using the `SoundSpec` level concatenation operator:

```haskell
concatSS = foldl (:+) NopS
```

**Noises and game start sounds**

The very easiest sounds to program for our DSL were the sounds that just needed to sound somewhere once. These consisted of a single sound file which was run all the way through.

All of these sounds were global. A typical sound definition for an environmental noise would look like the noise of a distant explosion.

```
Trigger "explosion distant 5"
  (MatchEvent "sound.trigger.explosion.distant.5")
  (Execute "explo distant 5"
    (concatSS
      (map
        (\i -> Play "Random/explosions/explo_distant_med_05" i 0 (100,100))
      global)))
```

**Listing 1:** Reactive event definition to play the sound of a distant explosion.

This sound would be triggered by the game masters by pushing a button in their sound control console that generated a `sound.trigger.explosion.distant.5` event on the joint AMQP communication bus.
The Monad.Reader

Trigger "celestra act 3 trigger"
(MatchEvent "sound.trigger.act3")
(Execute "celestra act3"
(concatSS
(map
  \i -> Play "Story/Celestra_Akt3" i 0 (100,100))
global)))

Listing 2: Reactive event definition to start the game session starting sound sequences.

The four different game starting sound sequences – with mood-setting music and a spoken recap – were triggered in the same way.

The same structure was used for game end sequences: depending on which of the six pre-written game ending sounds was deemed appropriate, a different trigger was sent over AMQP. This then phased over to an improvised game master epilogue spoken over the ship’s PA system.

Participant-triggered noises

Several of the participant actions were supposed to trigger noises as well. Whenever one of the players initiated a sensor scan or a torpedo loading procedure, sounds were triggered that corresponded to that soundscape. The only fundamental difference between these and the game-mastering sounds described above was that the triggers used were game system events. As an example, we can consider the active scanning sensor system.

Trigger "dradispingtrigger"
(MatchEvent "space.sensor.dradis.ping")
(Execute "dradisping"
(concatSS
(map
  \i -> Play "112_active_dradis" i 1000 (80,80) :
    Play "330_activating_dradis" i 0 (100,100))
global)),

Listing 3: The event definition for the sound system to react to a participant-triggered active scan ping
As a player successfully initiates an active scan, the sensor control console sends a `space.sensor.dradis.ping` message over AMQP. This is picked up both by the game simulation system, that reacts to the use of the active sensor in the simulation, and by the sound system that launches two sounds to be played on the entire ship. First (with time-delay 0), the sound `330_activating_dradis`: a synthetic voice announcing the activation of the dradis system, and then, a second later (time-delay 1000ms), the actual ping sound `112_active_dradis`.

**Figure 3:** The control panels for the active scanner and the FTL drive.

### Ambient loops

The game was designed with three ambient noise loops: one global, with the general soundscape of a space ship in action, and two localized loops with reactor sounds and the command bridge sounds. All these needed to be started as each game session started up, and we built a single trigger to create all of them:

The bridge was controlled by stations 8–11, while the large sub-woofer system for the deafening noise of the reactor engines was sound station 12.

In addition to these loops, one more looping sound was included. If a general alarm was sounded for some reason, the blaring siren sound sat in a loop construct.

### Attack-decay-sustain-release

A large family of the sounds used in the soundscape went through a sequence of sound files, with fades between them. First, one sound would start the sequence. Next, a loopable sound would keep the sequence running – preferably of a length that could be determined by the game masters at will. Finally, a sound would finish the noise sequence, often indicating success or failure of the corresponding player action.
Trigger "startup loops trigger"
(MatchEvent "sound.trigger.startup")
(Execute "bridge ambience loop"
(concatSS
(map
(i -> Loop "136_bridgeambience" i 0 (10,10))
[8,9,10,11])) :++
Execute "reactor ambience loop"
(Loop "124_reactor_run" 12 0 (60,60)) :++
Execute "sound of celestra loop"
(concatSS
(map
(i -> Loop "103_soundofcelestra" i 0 (20,20))
global))),

Listing 4: Reactive trigger that launches the ambient sound loops at the start of the game.

Trigger "alarm trigger"
(MatchEvent "sound.trigger.alarm")
(Execute "alarm"
(concatSS
(map
(i -> Loop "137_generalalarm" i 0 (50,50))
global))),

Listing 5: Reactive trigger that launches the general alarm siren sound as a loop.
Henrik Bäärnhielm, Daniel Sundström, Mikael Vejdemo-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP

These were common enough and repetitive enough that we ended up building a specific function to generate them. In the SoundSpec.hs source file, we defined a function `attackLoopDecay` that generated a sequence of chained sound commands.

First, we execute the sound `atF` at loudness `atL` and at all the locations in `locs`. Next, we launch the loop after a delay of `t` and at loudness 0. We launch our cross-fade sequence at `t+100ms` going from the attack sound to the loop sound. Finally, we generate a family of triggers for the possible phasing out options. Each such trigger has a numbered identifier, and contains both the required action for the phasing out option in `sc`, as well as commands to stop playing the loop and to clean out all the alternative phase out options.

With this utility function in place, we are able to construct sound sequences for potentially failing player-initiated events – such as using the Faster-Than-Light jump drive.

When the FTL command console initiates a jump sequence, the AMQP message `space.console.ftl_jump_started` is issued. This launches a sound system reaction that first triggers the synthetic voice announcing ship-wide that FTL has been initiated, by playing `314_initiating_FTL` everywhere.

Next, we call the `attackLoopDecay` construct to generate the sounds for spinning up and running the FTL engine. The related sounds are named `ftl jump attack`, `ftl jump loop`, `ftl jump xfade`, `ftl jump trigger 1` and `ftl jump trigger 2`.

The two triggerable end sounds are triggered with the game master generated `space.console.ftl_jump_completed` and `space.console.ftl_jump_failed` messages respectively, and play a sound and a synthetic voice message corresponding to the result.

Countdown

Never actually used in game, we also prepared a way to generate a synthetic voice counting down. We had pre-recorded numbers that could be concatenated, but needed to string them together programmatically.

To do this, we adapted the game engine to send out timing signals over AMQP every second, every 10 seconds, every 30 seconds and every minute. With these, we could construct functions that generated triggers listening to these timing signals and generating appropriate sound commands for the countdown sequence. They were all variants of the same fundamental structure, illustrated in Listing 8

Calling `minutes 3` would store a SoundCommand named `countdown 3m0s` that generated a trigger waiting for the next `sound.heartbeat.minute` event and then playing a synthetic voice saying `Countdown T minus three minutes and counting`. Finally, the event would look up and call the stored SoundCommand named `countdown 2m0s`. 

21
attackLoopDecay ::
  Id -> -- ^ Slug to generate ids for this command set
  Id -> -- ^ atF: attack sound
  Loudness -> -- ^ atL: attack loudness
  Int -> -- ^ t: time before fadeover
  Id -> -- ^ lpF: loop sound
  Loudness -> -- ^ lpL: loop loudness
  [Int] -> -- ^ locs: location list
  [(FilterSpec, SoundCommand)] -> -- ^ opts: Options for phasing out
  SoundCommand

attackLoopDecay slug atF atL t lpF lpL locs opts =
  Execute (slug ++ "attack")
    (foldl (:+) NopS (map (\i -> Play atF i 0 atL) locs)) :++
  Execute (slug ++ "loop")
    (foldl (:+) NopS (map (\i -> Play lpF i t (0,0)) locs)) :++
  Execute (slug ++ "xfade")
    ((FadeId (slug ++ "attack") t (t+100) (0,0)) :
      (FadeId (slug ++ "loop") t (t+100) lpL)) :++
  (foldl (:+++) Nop
    (map
      (\ (i,(fs,sc)) ->
        Trigger (slug++"trigger"++(show i))
        fs
        (sc :++
          foldl (:+++)
            (Execute (slug++"stop")
              (StopId (slug ++ "loop") 0))
            (map
              (\j -> Delete (slug ++ "trigger" ++ show j))
              ([1..length opts])))
      )
    (zip [1..] opts)))

Listing 6: The attackLoopDecay utility function
Trigger "ftl jump trigger"
(MatchEvent "space.console.ftl_jump_started")
((Execute "ftl jump voice"
  (concatSS
   (map
    (\i -> Play "314_initiating_FTL" i 0 (80,80))
global)) )++
(attackLoopDecay "ftl jump"
 "117_FTLspinningup" (80,80) 150
 "123_FTLspinupcomplete" (80,80)
global
 [(MatchEvent "space.console.ftl_jump_completed",
  Execute "ftl jump completed"
  (concatSS
   (map
    (\i -> Play "119_FTLjump" i 0 (80,80) :
     Play "318_FTLcomplete" i 9000 (100,100))
global))],
(MatchEvent "space.console.ftl_jump_failed",
 Execute "ftl jump failed"
 (concatSS
  (map
   (\i -> Play "118_FTLfail" i 0 (80,80) :
    Play "317_FTLmalfunction" i 4000 (100,100))
global)))))

Listing 7: Reactive trigger event to launch the FTL jump sound sequence, with two options for sequence finish: one for success, one for failure.
minutes n =
  Declare
    ("countdown " ++ show n ++ "m0s")
    (Trigger "countdown"
        (MatchEvent "sound.heartbeat.minute")
        (Execute "countdown"
            (concatSS
                (map
                    (\i ->
                        Play "Countdown/T minus" i 0 (100,100) :
                        Play ("Countdown/" ++ show n) i 1000 (100,100) :
                        Play "Countdown/Minutes and counting" i 2000 (100,100) global)) ++
                    Call ("countdown " ++ show (n-1) ++ "m0s")))

Listing 8: Function to parametrically define a trigger event to launch a minute-by-minute count down sound sequence.

For the last 15 minutes, the countdown would drop down to counting every 30 seconds; for the last minute, it would count every 10 seconds, for the last 10 seconds, it would count every second.

Access to these chains of callable commands was given to the game masters by generating triggers through versions of the trigger given in Listing 9.

minuteTrigger n =
  Trigger
    ("countdown trigger " ++ show n ++ "m0s")
    (MatchEvent ("sound.trigger.countdown." ++ show n ++ "m0s"))
    (Call ("countdown " ++ show n ++ "m0s"))

Listing 9: Game master callable triggers to launch the countdown sequence from a parametrizable starting time.

When an AMQP message of the form sound.trigger.countdown.25m0s arrives, the trigger generated by a call to `minuteTrigger 25` is activated. This trigger looks up and calls the stored SoundCommand named countdown 25m0s, which hooks into the chains of SoundCommands described above.
Experiences

The experiences gathered in this project can be organized into the various stages of the project.

Rapid development and package designs

It turns out that the amqp package expects lazy bytestrings and the hedis package expects strict bytestrings. The bytestring package installed had no automatic way to convert between these – and it took a while in the project to figure out how to make all systems interoperate.

Several features of the resulting sound scape daemon were direct consequences of package and platform choices. The automatically generated serialization and parsing capabilities, both in the Show/Read dyad and in the Generic generation of JSON and bytestring serializers all mean that parsing and serialization worked automatically and immediately. Using the callback structure of the amqp library also meant that a reactive server loop was provided essentially for free.

All these features contributed to a rapid development with large amounts of functionality arriving early in the process.

Interfacing to other component platforms

Since the entire project was communicating by AMQP with JSON-encoded packages already, the use of these standards made interoperability easy. One of the most difficult parts of this side of the development process was in inspecting emitted JSON packages to figure out the particular encoding that the Generic-generated JSON parser worked with, so that appropriate and parseable structures could be emitted from the Ruby control interfaces.

Once the reactive interfaces through the amqp module were in place, it quickly became clear to us that the easiest way to provide user interfaces to the daemon would be through a Ruby web application front end that emitted packages crafted to trigger specific rules in our trigger system.

Interfacing to creative design teams

One original expectation that did not work out as expected was to produce a language for other users to work with; as often is the case with large scale projects relying on voluntary work and expansive creative vision, work on the project progressed beyond the launch of the first performance. The sound design teams worked on sound design far beyond the stage where they would have needed to familiarize
themselves with syntax and functionality of the sound specification system to use it themselves.

Instead, as a compromise, we requested all relevant information to encode the designed soundscapes; and iterated through requests for more information and educated guesses until soundscapes were defined and could be tested. Here, the choice of embedding the language within Haskell paid off; the sound specification programmer was able to use many Haskell-specific idioms and structures to produce long sequences of parametrized and repetitive soundscapes definitions. These were in the end stored as commands stored in triggers, listening to specific triggering packages, often creating new triggers for later reactions. This enabled constructions where a player action (pressing the Load Torpedos button) would start some sounds (a loading torpedo noise), but then wait for game master confirmations before playing further sounds (a loading-finished noise and a robotic voice confirmation). The later sounds would be temporarily installed trigger definitions waiting for a confirmation package and including cleanup code that erased all the temporary trigger conditions after sound execution, as described in the examples above.

**Latency in performance**

There were latency issues in the performance use. We had, in the end, 320 trigger expressions, each carrying up to 300 sound play events in short sequences. At the first performance, the on-site crew noticed latencies reaching up towards 8–10 seconds between triggering action and sound execution. Before we were able to isolate the cause of these issues, and while we were still comparing complexity properties of the chosen containers – Haskell lists and strict hashmaps depending on whether inspection of all members would be a common operation or not – the performance issues decreased to within a few seconds. Given the acoustic layout of the ship, this was deemed sufficient for further performances.

It should be noted that the high-latency results required some dramaturgic edits; certain sound effects were not used because they were dependent on low latency, and thus infeasible even with the reduced performance issues. It was decided between the first and the second performance to run with the system as built and not try for specific further optimizations.

**Hardware Issues**

An issue arose based on the hardware. Once the system was properly deployed and the project entered the testing and verification stage, it was discovered that before playing sounds, the system emitted a few seconds of pops and noise regardless of the sound played. After researching the issue, this was proven to be a known issue with the RP platform, stemming from the way power supply is routed to the
Experiences on-site

The system was installed in the war ship two days before start of the game. Installation was done in three steps; preparing clients, server-side, and on-site configuration. Since circumstances at the location were more spartan than other, off-site locations available to the team – most preparation was done in-office and then transported to the actual game location.

Client preparation consisted of pre-loading SD cards for the Raspberry Pis with client daemons, the pre-configured sound library and connection bridges. A disk image was prepared and then duplicated onto the memory cards. This saved time at installation time, but proved to be a bad choice on-site. Certain updates could be performed over the network – but due to issues with the disk image duplication the memory cards had to be reprogrammed. This could only be done with the physical cards in hand and since the Raspberry Pis were distributed over a large area and, in many cases, in locations hard or cumbersome to reach – this proved to be a larger job than anticipated. Apart from this, pre-loading the memory cards worked very well.

Server-side installation and configuration was done via cabal on the main game server. Haskell code was kept up-to-date via a Git repo and regularly synchronized from an online repository. Installation on server went as expected and was very easy to get set up.

On-site configuration was designed to be done in three steps; physical installation, soundscape configuration and verification. Physical installation was cumbersome but not beyond expectations. Soundscape configuration worked almost as expected. The Commit command built into the system proved invaluable since it allowed for one part of team working remotely to write an import-definition of the soundscape then send it to the on-site team, who imported it into the server and then made local backups and exports.

Conclusion

Haskell as a platform, coupled with programming techniques connected to a functional reactive programming style, datatype construction for domain modeling and a package collection that neatly covered all required technological dependencies all combined to a successful project where the sound specification system produced significant value to the game experience and to the immersion into the game world.
for the participants. Features of Haskell as a platform and of the packages used produced a more capable deliverable than the original specification had expected.

We are not releasing the source code for this system, however, we encourage anyone interested in further details or derivative projects to get in touch with any of the authors.

References


Henrik Bäärnhielm, Daniel Sundström, Mikael Vejdemos-Johansson: A Haskell sound specification DSL: Ludic support and deep immersion in Nordic technology-supported LARP


