Chime: Anonymous Channel Based Messaging for Smart TVs

Marc Leef
Princeton University
88 College Road West
Princeton, NJ 08540
+1(503) 750-1809
mleef@princeton.edu

Hussein Nagree
Princeton University
88 College Road West
Princeton, NJ 08540
+1(734) 604-3523
hnagree@princeton.edu

Abstract

Increasing television interactivity is one of the key challenges facing the television industry. As of now, this is mainly achieved by integrating third party services, such as Twitter, into television feeds. However, this method is cumbersome for two reasons – it requires users to maintain accounts with these external services and it forces users to utilize an additional device, such as a phone or laptop, while watching TV in order to share their thoughts. In addition, using a third party app such as Twitter doesn't allow for anonymity, a desired feature in many large scale chat based systems. This paper presents the design and implementation of Chime (https://github.com/mleef/Chime), an anonymous, real-time, channel based messaging platform for Smart TVs. Chime treats each television channel as its own chat room to enable interactive television viewing. Users send and receive messages to all other users of the system who are currently watching the same channel and the messages are displayed at the bottom of the screen in a ticker style format. Chime is extremely versatile and lightweight, allowing it to be easily deployed on a number of different host environments and facilitate communication across multiple Smart TV operating systems. It uniquely leverages Smart TV platforms by combining messaging and entertainment into one seamless experience, eliminating the need for additional devices or services as seen previously.

1. Introduction

In recent years, televised media programs have popularized a ticker format for rapid news delivery, as seen in Figure 1. This ticker encompasses a small section somewhere on the screen where new bits of information, either closely related or completely tangential to the primary content, become visible to viewers, usually via slowly scrolling text. On ESPN, for instance, the ticker may contain game scores, player statistics, and social media postings relevant to the current program. On CNN, tickers may emerge to deliver breaking news, provide background information on the current story, or advertise future programming. Financial programs may use a ticker to provide up to date information on fluctuations in individual stock prices. In general, tickers provide a flexible way to augment the viewing experience with additional information without interrupting the primary programming.

1.1 An Underutilized Platform

While Smart TVs provide a unique platform on which to develop applications, we feel they have been sorely underutilized. A high-level analysis of the most popular Smart TV apps reveals that most applications do not take advantage of the fact that they are running on a television. While there are quality weather, game, and news

Figure 1: Current example of ticker usage to promote interactivity in television

applications, these are platform agnostic apps that provide a similar user experience on other devices and do not take advantage of the unique Smart TV platform. Streaming applications are a key exception, as they provide access to services directly in-line with the typical television use-case: consuming visual entertainment. These apps, published by companies like HBO, Amazon, Netflix, ESPN, etc. dominate the Smart TV 'Most Popular' charts.

1.2 Television Interactivity

Making the TV experience more interactive is a sensitive issue. On one hand, if users want to increase the interactivity of their viewing experience there are multiple avenues to do so: watching with friends, participating in online forums, posting on social media, etc. Conversely, if users want to enjoy their programming in isolation, forcing interactivity can be extremely detrimental to the viewing experience. Some modern television networks attempt to strike a balance of interactivity by flashing program-specific hashtags, websites, and other content during the broadcast, allowing viewers to engage further if they wish to do so while not alienating those who do not. Because the last thing networks want to do is alienate their viewership, novel efforts to enhance the viewing experience have been relatively non-existent and lackluster.

1.3 Overview

Chime aims to quash both the aforementioned problems. By creating a chatroom-like environment for each channel, we're leveraging a television-specific feature on which to base our system. In addition, Chime eliminates the need for additional devices or services, thereby increasing the interactivity of watching television while still allowing for an effortless and passive viewing experience. Our first version of Chime was implemented as a foreground app for the Samsung Tizen Smart TV operating system,

as well as a YouTube-channel based website. The backend was deployed to a single Amazon EC2 instance.

The paper is laid out as follows: Section 2 presents the system architecture and detailed design of the backend and frontend designs. Section 3 evaluates our performance on some sample use case scenarios and discusses how the system would scale. Section 4 covers related work and Section 5 discusses the implications of our work.

2. System Architecture

2.1 Backend Design

Our prototype implementation of the Chime backend is monolithic and difficult to scale, but is otherwise quite functional and representative of the capabilities of the full system. It contains a number of different services that can be accessed via various ports (socket/web socket connections, HTTP requests, etc.) on the same machine. Our more mature, revised design splits the various pieces of functionality provided by the prototype across multiple machines and consists of a number of master/worker relationships enabled by a simple communication layer. Both versions were written in Java. In this section, we provide an overview of the most important aspects of our system design, and highlight the key differences between our prototype and final systems.

2.1.1 Mappings

To maintain a consistent view of the current global state of the system, which in this context is comprised of the relationships between the various channels, televisions, and sockets, we utilized a number of concurrently modifiable hash maps. These data structures needed to provide both thread-safety, due to the high thread count necessary to manage many client connections simultaneously, and fast look-up times, due to the real-time and potentially large-scale nature of the system. The primary mappings the system maintains are channels to watching televisions (*ChannelMap*), televisions to sockets/web sockets (*TelevisionMap* and *TelevisionWSMap*), and a reverse mapping from sockets and web sockets back to televisions (*SocketMap* and *WebSocketMap*).

A visualization of these relationships is provided in Figure 2. The channel map is updated when clients change channels, while the television and socket maps are updated when clients open or close a connection. Java's provided implementation of concurrently modifiable hash maps uses locks to secure key ranges that are being written to and allows reads to be executed without locks, making it perfectly suited for the performance requirements of our system. In our final design, the master node is responsible for maintaining the channel map as well as an additional new structure, the worker map, which links worker nodes to the televisions/sockets maintained by the given worker. Worker nodes retain their local television and socket mappings and simply propagate channel changes up to the master's channel map.

2.1.2 Managers

Our architecture consists of a flat hierarchy of managers, classes that are responsible for a narrow set of related tasks. These task groupings consist of networking actions such as the handling of socket connection events (*ChimeSocketManager*), web socket connection events (*ChimeWebSocketManager*), and RESTful style

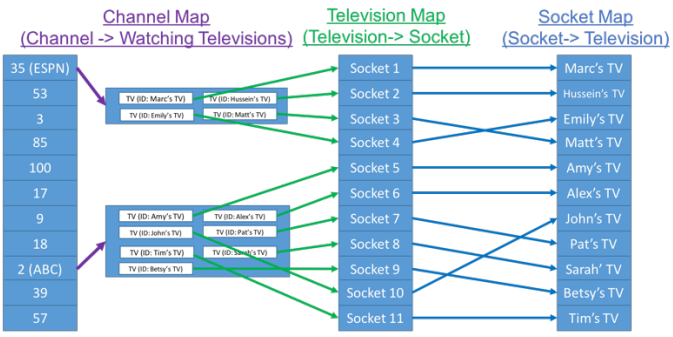

Figure 2. An overview of the various relational information maintained by the system in concurrently modifiable maps.

HTTP request event handling (*ChimeRestManager*). Other groupings involve maintaining system consistency through propagating updates to the previously discussed mappings (*MapManager*) and performance related tasks such as handling the intermittent garbage collection of unused television and socket objects (*CleanupManager*). The primary driver of our prototype implementation is the *ServerManager* class, which initializes instances of the aforementioned managers and maps before beginning execution. The *ChimeSocketManager* makes heavy use of Java's provided non-blocking IO functionality for reading from and writing to sockets. An early iteration of this manager utilized blocking IO to interface with sockets, but we quickly found this approach impractical as it required a one to one relationship of threads to client connections. Java's Selector class enabled us to maintain all connections using only a single thread. Additionally, this allowed us to batch the handling of newly generated IO events (sockets opened, written to, closed, etc.) together, further reducing the degree of multi-threading. *ChimeWebSocketManager* and *ChimeRestManager* make use of third-party libraries which afforded us much less concurrency control, but performed acceptably during our tests. Lastly, *MapManager* simplifies updates to the various mappings by abstracting away some of the lower level map manipulations required to maintain consistency.

Our final design contains two additional managers, *ChimeMasterManager* and *ChimeWorkerManager*. These classes are amalgamations of various pieces of functionality from other managers that have been repurposed to achieve the desired distributed behavior. *ChimeWorkerManager* instances both listen for and send HTTP requests to communicate with the master, as well as handling new client connections. *ChimeMasterManager* instances have similar HTTP communication infrastructure and maintain consistent global views of the workers and clients.

2.1.3 Messaging

We used lightweight JSON messages to facilitate communication between both client and server in our prototype and master and worker in our revised design. Various message types include registrations (channel change, new connection, new worker events), chimes (message to be broadcasted to other watching clients), and a variety of response and status types. We employed the use of Gson, Google's open source JSON manipulation library, to handle the serialization between Java object instances and raw JSON. The actual sending of these various messages is handled by two networking focused classes, *HttpMessageSender* and *SocketMessageSender*. As the names imply, the classes provide functionality for sending HTTP requests and writing data to sockets

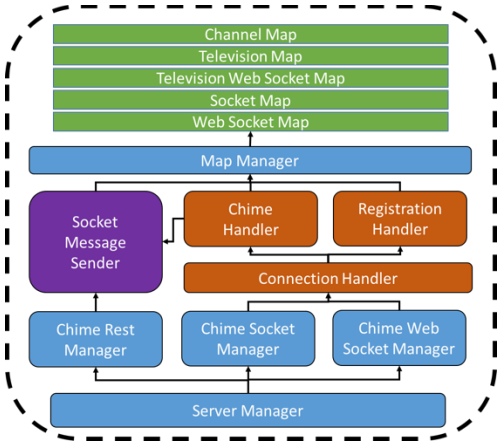

Figure 3. Workflow of our prototype implementation. This monolithic system is capable of of executing all necessary procedures at the cost of single machine limitations.

respectively. These classes also leverage the *MapManaager* to maintain consistency in the case of errors (closed sockets, error response codes, etc.). The prototype implementation doesn't make use of *HttpMessageSender* at all, as it primarily listens for HTTP requests but doesn't make any. HTTP messages are crucial to the successful operation of the master/worker paradigm, as the various nodes are in constant communication, forwarding client messages and managing mapping updates.

2.2 Flow of Execution

2.2.1 Prototype

Our prototype implementation is able to perform all necessary system tasks but suffers from a few crucial limitations due to its single node, monolithic design. Upon startup, the *ServerManager* initializes all other managers and data structures, before starting execution by dispatching threads in which each manager begins listening on its own port. New clients open sockets on the relevant port (4444 for sockets, 4445 for web sockets) which are stored in the appropriate map. These sockets remain open for the duration of the client's viewing session. Any messages written to the sockets by clients are collected into lists and processed in batches by instances of the *ConnectionHandler* class. This handler validates and classifies messages before dispatching appropriate sub-handlers (also in batches). In the case of a registration message, the MapManager is used to update the *ChannelMap*. For chime messages, the watching televisions on the associated channel are acquired from the *ChannelMap* and each of these television's socket type is determined (using the relevant maps) and written to with the content of the message by the *SocketMessageSender* class. If any of these writes fail, the sender will use the MapManager to update the television, socket, and channel mappings accordingly.

A visualization of this workflow is presented in Figure 3. While this system exhibited good performance and functionality, it was constrained by the number of concurrent TCP connections that can be maintained by a single machine. On the Amazon EC2 instance we experimented with, this number appeared to be around 10,000. As such, we decided that a division of responsibility across multiple nodes was necessary to overcome these physical limitations and scale to hundreds of thousands to millions of concurrent users.

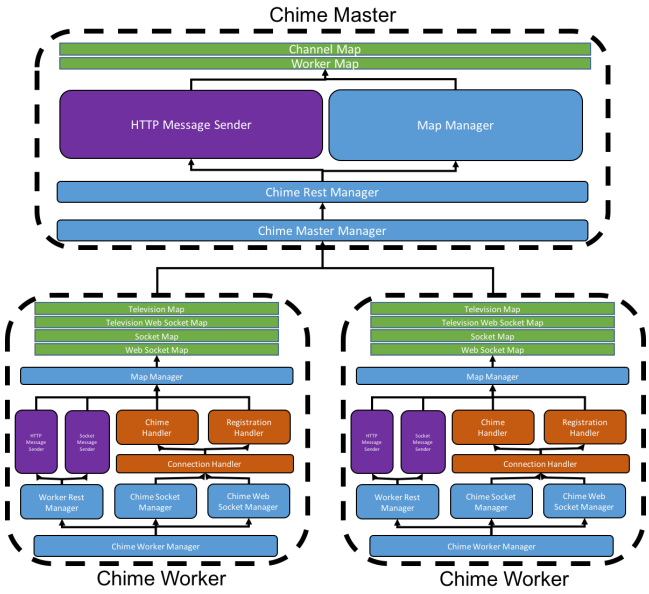

Figure 4. Workflow of our final implementation. The most significant deviation from the prototype lies in the distribution of responsibilities (sockets, maps, etc.) across multiple nodes.

2.2.2 Final Design

Our revised system design sports a hierarchical division of duties between a single master node and a load balanced set of worker nodes. The *ChimeMaster* class is responsible for maintaining a globally consistent *ChannelMap* and for directing clients to *ChimeWorker* instances that have the capacity for additional socket connections. In this workflow, clients contact the *ChimeMaster* to obtain the URL of a *ChimeWorker* with which to open a connection. Our implementation contains a naïve load balancer that simply directs clients to workers until the given worker is at capacity. Instances of the *ChimeWorker* class are very similar to the monolithic prototype in their ability to accept and maintain a number of socket connections from clients, but also have the additional functionality to send and receive messages from the *ChimeMaster* using HTTP.

In this design, clients still write registration messages to their associated socket, and these changes are then propagated to the *ChimeMaster* by the respective *ChimeWorker*. Chime messages are similarly forwarded to the master through individual worker sockets. The master, using the global *ChannelMap* and *WorkerMap* (a mapping from worker URLs to associated televisions/sockets) first determines the watching televisions on the channel associated with the chime. It then determines which workers contain each of these televisions before sending HTTP requests to the respective nodes with the chime message. Upon receiving these requests from the master, the workers write these chimes to the relevant sockets. Despite the somewhat convoluted nature of this workflow, it allows *ChimeWorker* instances to scale easily as the number of clients increases. Only one *ChimeMaster* is required, because maintaining the mappings in memory is quite lightweight and the HTTP request handling is parallelized. Additionally, both master and worker instances possess shut down hooks which notify other nodes in the system in case of a crash to maintain hierarchical consistency.

2.3 Frontend Design

The frontend of Chime was implemented in two phases – first as a demonstrable, proof-of-concept design as a website and then as an app for the Tizen Smart TV operating system. We further elaborate on the two frontend designs below.

2.3.1 Chime Web Implementation

The first implementation of Chime was hosted on the following webpage: http://www.cs.princeton.edu/~mleef/static/chime-demo/ This implementation, as seen in Figure 5, was made for a version 0 demo, and consisted of a number of YouTube videos, each acting as their own channel. The YouTube video plays in the center of the screen. Below it, a channel toggle allows users to switch between available channels, and a text entry box enables sending messages. The messages are displayed as lines of text that move across the screen from right to left, with newer messages appearing below each previous message. Any number of users can visit the website, which displays a count of the number of visitors currently viewing a particular channel.

The page makes a web socket connection with our EC2 web server, and then keeps this socket alive over a persistent TCP connection. When the page loads, a registration message is sent to the backend over this web socket, informing the backend of the current channel the user is watching and the user's television ID (which is generated as a random alphanumeric string for each visitor to the page). Clicking the Submit button sends a chime message to the backend using the web socket, with information about the text entered, the current channel and time, and the television ID. Changing the channels, sends a new registration message over the web socket, with fields for the previous channel, new channel, and television ID. The persistent TCP connection is crucial for receiving incoming messages. While the socket is open, it listens for new messages, and on receiving a message, it appends the message contents onto the ticker div of the HTML page, thereby displaying it on a new line of the page.

This implementation was built as a straightforward way for multiple people to interact with the Chime backend, as well as demonstrate its versatility in interacting with a number of different frontend applications. Since the only unique identifier (besides the basic information shared over the web socket) is a randomly generated string ID, the user is effectively anonymous to the server (an IP address masker can be used by a user that wants to further hide information that may be conveyed through the web socket). However, there were a number of features that we did not implement in this version of our frontend (such as correctness and ordering guarantees), as we decided to focus our additional efforts on seeing if we could replicate the results in a Smart TV app.

2.3.2 Chime Tizen TV Implementation

The flexibility of the backend allows the frontend application to be implemented on a number of Smart TV operating systems, such as Android TV, LG's WebOS, or Samsung's Tizen. This can be done with a minimum overhead, and each system will be instantly compatible with the other due to the uniform handling of web sockets and sockets by the backend. We decided to build our initial app on the Samsung Tizen Smart TV operating system.

The app is based heavily on the web implementation, and inherits the same basic behavior. A web socket is created on load, which sends a registration message, with the television's IP address as its ID. Channel change event listeners also send registration messages

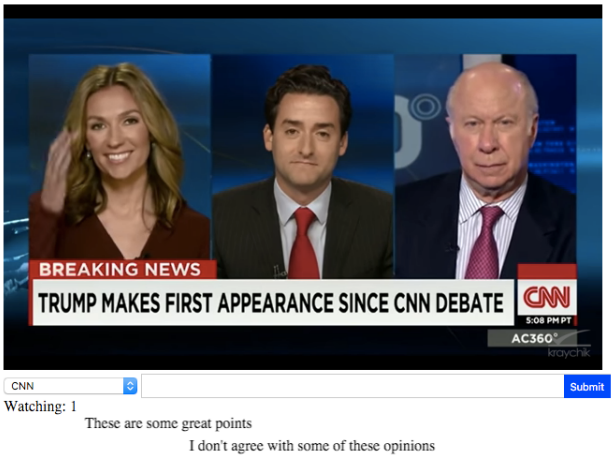

Figure 5: Screenshot of webpage executing Chime instance

when a channel change is detected. Chime messages are sent individually by the app, and the web socket listens for incoming chimes and displays the message content on the screen. The web socket remains open as long as the app is open or the TV is on, thus allowing the TV to constantly listen for incoming chimes, as well as send new registration or chime messages without reestablishing the TCP socket connection.

The first change we made from the original implementation was related to the way channel changes are handled. Previously, a channel change would immediately fire a registration message. Here, we first wait for a couple of seconds, assert that the channel changed to when the channel change listener was called is the same channel the TV is currently tuned into, and send a registration message if that is true. This is because we want to avoid sending lots of registration messages if a user is rapidly flipping through channels – that way the user doesn't get half messages for channels they're not interested in watching, and the backend doesn't get bogged down unnecessarily updating the channel mappings. The second major change relates to error checking. Since registration message sending is delayed by the frontend, the web socket may continue to receive chimes from previous channels. In order to avoid this problem, the channel info of incoming chimes is validated before displaying the message content. We also added in additional logic to discard old chimes that were sent significantly before the current time, with the exact value determined to be 'significant' found subjectively.

We attempted to implement a number of additional features, such as voice recognition, into the app. Our goal initially was to have the user interact with the app entirely through voice. Unfortunately, without an actual Samsung Tizen TV for testing, we were unable to test the voice performance in a real world system outside of the emulator. Similarly, working with actual television channels would not be possible on the emulator alone. Ultimately, given that our Chime servers were able to communicate effectively with our Chime app on the emulator, we found that proving correctness and testing performance could be reasonably done with the current version of the app. Thus, we decided against spending more time implementing app features, as without a TV for final testing it was unlikely that we would be able to release this app to the public. Based on our experiences (and difficulties) developing for the Tizen platform, we feel continued work on a client-side Chime application would probably be better suited for a different platform, such as Sony or Android TV.

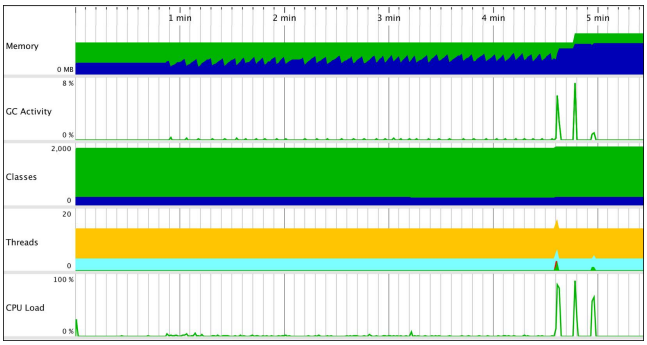

Figure 6: Measuring CPU load and memory usage of the server during a simulated test with 5000 connections.

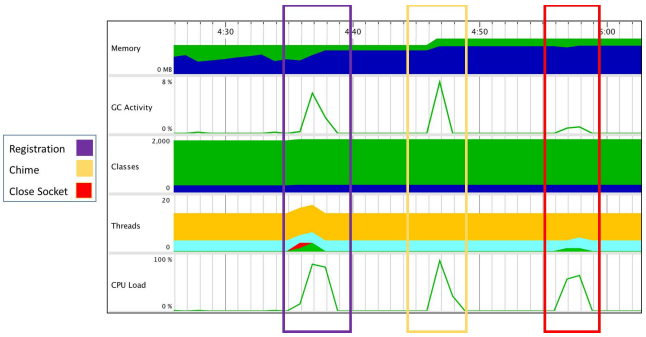

Figure 7: A zoomed-in look at the periods of heavy activity from Figure 6.

3. Performance Evaluation

In order to test performance, we set up a number of Python scripts to establish socket connections with the server, and then send registration and chime messages over those sockets. These sockets would disconnect all at once, which resulted in the server's garbage collector finding all the closed sockets and removing the elements from the mappings accordingly. Ideally, we intended to set up a real world test for our server with 100,000 to 1,000,000 simultaneous connections, but the Amazon EC2 instance that the Chime backend runs on limits the total bandwidth and CPU available to the server. Instead, we ran our tests with 5,000 connections (close to the maximum allowed by Amazon for our basic plan) and tried to effectively extrapolate our results for a system that would need to handle millions of simultaneous connections.

Figure 6 shows the memory usage, CPU load, and thread creation in the server when a script establishes 5,000 distinct connections. The sockets are created one by one over the span of a few minutes, which can be seen in the saw-tooth-like memory graph initially. This is followed by a rapid sending of registration messages, a pause, a rapid sending of chime messages, another pause, and then an exit from the script (which disconnects all the sockets). Each of these correspond to the spikes in CPU load in Figure 6. Responses from the server to the script were near instantaneous, and the total time taken was mostly dependent on network connection speed and the sequential nature of the scripts.

Figure 7 examines each of the spikes in more detail. It is visible that while each spike corresponds to a fairly high CPU load, the total number of threads created is quite low due to the grouping of thread tasks. This is encouraging as even if the tests were to be scaled up to millions of concurrent connections, our system would be able to effectively respond to these connections without spawning an unreasonably large number of concurrent requests. Since each request requires no more than two map lookups from the Chime master server, requests are handled quickly with very little bottlenecking at the server. In our tests, the additional memory consumed is fairly low (under 10 MB – exact units were cut off in the Figure 7) even at 5000 connections, which enables the entire maps to be stored in memory, enabling quicker lookup. Due to the lightweight setup of the server, which uses 32 bit IP addresses and 32 bit identifiers for channel numbers and socket worker IDs, each additional connection uses 8 bytes in the socket map, 8 bytes in the channel map, and 8 bytes in the television map. Extrapolating from require less than 2 GB of memory, allowing it be wholly contained in RAM.

With the addition of the distributed connection handling, our Chime backend has very little bottlenecking or load balancing issues. We considered distributing our channel and television mapsthe maps are expected to fit in memory even upon scaling, which meant that distributing the system would only effectively reduce one map lookup in the master and increase overall response time. Overall, the performance of the system was quite satisfactory, as the time taken to display the message in the client is always greater than the next message response time. This creates the perception of a smooth, continuous stream of messages for a user of Chime.

As previously discussed, our final design further expands upon the performant prototype by shifting the channel mapping and chime message delegation responsibilities to a single master node while a set of load balanced worker nodes maintain socket connections. In this way, the workers can be scaled with the number of concurrent viewers to achieve high performance in the face of per-machine socket constraints.

4. Related Work

While work in this specific area is limited, we were able to find a few relevant sources. Luyten et al. explore techniques to increase the interactivity of the television experience by using web applications to augment the screen with avenues for viewers to communicate with one another. Wohn has also conducted work in this space, by evaluating the use of Twitter as a medium to share television viewing experiences and analyzing the progression of socialization techniques throughout the history of the television. While these projects share similar goals with ours, they don't focus on building a real-world system to enable these interactions.

5. Conclusion

Smart TVs are becoming increasing popular, and there is a demand to make use of the unique nature of this platform to create useful and novel applications. Chime is an effort to address this need, by introducing optional interactivity to the previously exclusively passive experience of watching television. The system is lightweight, versatile, and easily scalable, which make it perfectly suited to the current Smart TV market. Chime has been tested with both real-world and synthetic workloads to ensure that it is robust and responsive. While there are a few features missing before it is market ready, Chime brings some much needed innovation to the current crop of television apps.

6. References

[1] Gosling, J. (2000). *The Java language specification*. Addison-Wesley Professional.

[2] Luyten, K., Thys, K., Huypens, S., & Coninx, K. (2006, April). Telebuddies: social stitching with interactive television. In *CHI'06 extended abstracts on Human factors in computing systems* (pp. 1049-1054). ACM.

[3] Samsung Smart TV Apps Developer Forum - SDK download, Guide & Forum. (n.d.). Retrieved December 31, 2015, from http://www.samsungdforum.com

[4] Wohn, D. Y., & Na, E. K. (2011). Tweeting about TV: Sharing television viewing experiences via social media message streams. *First Monday*, *16*(3).

[5] Wohn, Y. (2013). History of Social Television.