Technische Universität Berlin

Institut für Telekommunikationssysteme
Fachgebiet für Quality Engineering of Open Distributed Systems

Fakultät IV
Einsteinufer 25
10587 Berlin
http://www.qds.tu-berlin.de/

Master Thesis

Design and Implementation of a Development and Test Automation Platform for HbbTV

Christian Bromann

Matriculation Number: 359957
16.08.2017

Supervised by
Prof. Dr.-Ing. Ina Schieferdecker

Assistant Supervisor
Dipl.-Ing. Louay Bassbouss

FOKUS Institute
Kaiserin-Augusta-Allee 31
10589 Berlin

This dissertation originated in cooperation with the Fraunhofer Institute for Open Communication Systems (FOKUS).

I would like to thank Dipl.-Ing. Louay Bassbouss at the Fraunhofer Institute FOKUS for giving me the opportunity to carry out state of the art research in this field as well as providing me with helpful feedback at all times. It was really great to be able to wrap up my studies researching in one of my most passionate topics.

I also want to thank the whole team at FAME for being so wonderful human beings. I really enjoyed working together with you guys.

In addition, I thank all of my proof-readers for their work and their patience in looking through all chapters. Thank you Thomas Fett and Esben Baden Smith.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenständige Anfertigung versichert an Eides statt:

Berlin, den 14. August 2017

.....
(Unterschrift Christian Bromann)

Abstract

Hybrid Broadcast Broadband TV is one of the latest big developments in the TV industry. It is an effort to standardize the delivery of user-friendly enhanced TV services to the end consumer through connected Smart TVs and set-top boxes. As the standard evolves, it gets rolled out to more countries in the world. With more devices being equipped with this technology, a larger audience is getting access to it which opens interesting opportunities for broadcasters to create new revenue streams via advertisement or pay-TV platforms. Due to the increasing number of manufactures and device models in the market the support level is highly fragmented and aggravates the development of HbbTV applications. To ensure the functionality on a majority of devices a cumbersome and manual testing process is required which is opposed to current standards in software development. The software industry has shifted over the last 5 years from a milestone oriented to an agile approach where shipping qualitative software fast and iteratively is the number one principle. To establish a high development velocity a key factor for success is a solid continuous delivery pipeline that tests software in an automated fashion and provides confidence in the quality of the product.

The major objective of this study is to improve the process of building HbbTV applications by implementing a developing and automation platform that helps HbbTV developers to overcome issues that have been around on web and mobile platforms for years. It examines current standards in debugging and testing of software applications and demonstrates how applying these best practices to the TV space helps to increase the development velocity and software quality of HbbTV applications. By building a debugging bridge that supports the Chrome DevTools Protocol it allows developers for the first time to inspect HbbTV applications in-depth and live on the TV using modern web authoring tools like Chrome DevTools. Furthermore shows this thesis how an Appium automation driver can use this bridge to run functional tests on real Smart TVs in an automated fashion based on the WebDriver protocol.

With that technology in place results demonstrate how powerful debugging HbbTV applications can be and how much information a developer can receive. From the full DOM tree to a JavaScript console up to a detailed report about all received network packages, the state of the app is accessible at any given point in time. Moreover, proves the new HbbTV driver that running automated tests in a continuous delivery pipeline using a tool like Jenkins is fast, reliable and interoperable. A comparison at the end reveals that this approach provides a higher scaleability, functionality and flexibility for debugging and automated testing than any other existing solution in the market before.

Zusammenfassung

Hybrid Broadcast Broadband TV ist einer der letzten größeren Entwicklungen auf dem Fernsehmarkt. Es ist ein Versuch, über die mit dem Internet verbundenen Fernsehgeräte oder Set-Top Boxen, benutzerfreundliche und erweiternde Sendeangebot an den Zuschauer zu vermitteln. Während sich der Standard weiterentwickelt, wird er bereits in immer mehr Ländern in der Welt bereitgestellt. So erhalten mehr Zuschauer Zugang zu den dargebotenen Services, die für Fernsehsender neue Vertriebsmöglichkeiten, wie Werbung oder Bezahlfernsehen, bieten. Wegen der großen Menge an TV-Herstellern und Modellen ist die Unterstützung der Geräte für HbbTV jedoch sehr unterschiedlich, was die Entwicklung von derartigen Anwendungen deutlich erschwert. Um sicherzustellen, dass diese dennoch auf den meisten Geräten funktionieren, ist häufig ein mühsehliger und manueller Testprozess notwendig. Dies ist mit heutigen Entwicklungsstandards allerdings nicht mehr vereinnehmbar. Die Software Industrie hat sich über die letzten 5 Jahre von einem wasserfall-getriebenen Modell zu einem mehr agilen Anzatz bewegt, bei der kontinuierliche Verbesserungen in kleinen Schritten eines der obersten Prinzipien ist.

Das Hauptziel dieser Master Arbeit ist es, den Entwicklungs- und Testprozess von HbbTV Applikationen zu verbessern, indem eine Plattform geschaffen wird, die HbbTV Entwicklern hilft, die selben Probleme, die es bereits für Web- und mobile Anwendungen seit Jahren gibt, zu lösen. Sie untersucht die aktuellen Standards in der Entwicklung und Qualitätssicherung von Software und zeigt auf, wie diese Methoden ebenfalls im Bereich des Fernsehens angewendet werden können. Indem ein Kommunikationskanal zwischen Applikation und modernen Entwicklungsanwendungen, wie z.B. den Chrome DevTools, geschaffen wird, ermöglicht diese Arbeit Entwicklern zum ersten Mal, detailierte Informationen über die Anwendung während der Entwicklung in Erfahrung zu bringen. Zudem zeigt sie auf, wie durch die Bereitstellung eines Automatisierungsprogammes, welches diesen Kommunikationskanal nutzt, funktionale Tests auf echten Fernsehgeräten ausgeführt werden können.

Durch diese Technologien wird demonstriert, wie effizient die Entwicklung von HbbTV Anwendungen sein kann. Der Entwickler erhählt, neben dem Applikationsaufbau, einer JavaScript Konsole und einer detailierten Auflistung aller Netzwerk Pakete, nicht nur umfangreiche Information über die Applikation an sich, sondern auch über ihren Zustand zu jedem beliebigen Zeitpunkt. Zudem beweist das Automatisierungsprogramm wie schnell, zuverlässig und vielseitig funktionale Tests in einem automatisierten Prozess, z.B. durch die Nutzung von Programmen wie Jenkins, ausgeführt werden können. Im Vergleich am Ende wird bewiesen, dass dieser Ansatz für die Entwicklung von HbbTV Anwendungen weitaus besser skalierbar, funktionaler und flexibler ist als jegliche bereits existierende Lösung auf dem Markt.

Contents

1 Introduction

In the late 19th century a German student named Paul Gottlieb Nipkow developed an electronical device that was able to send images over the wire with the help of a rotating metal disk. This was one of the first mechanical prototypes that suppose to become the television. With the beginning of the 20th century two types of TVs emerged: mechanical and electronic televisions. Whereas the mechanical type has seen a lot of innovation by 1934, all television systems had been converted to electronic machines. These became more and more popular in humans households so that almost 10 years later the number of U.S. homes with television sets could be measured in the thousands and by the late 1990s more than 98% of U.S. homes had at least one television. The TV was one of the first electronic devices that introduced a new generation of entertainment equipment that everyone was supposed to have.

Despite innovations like colored displays or flat screen devices the television quickly lost the race against the mobile phone as well as the personal computer and later laptop. Although it is still a very popular medium, people nowadays use their smartphone or personal computer more frequent as it was at the end of the 20th century. One of the major reasons was the innovation of the internet. The more people were able to connect to each other and consume media over the web the more became the television the less frequent choice for entertainment. The new generation of kids grows up with web media portals like YouTube[1] that generates billions and billions of clicks, views and revenue for advertisement companies every day.

Thanks to the most recent big innovations in television technology that are setup boxes and standards like Hybrid Broadcast Broadband (short for HbbTV) the TV industry tries to keep up with other technologies by introducing the so called smart televisions or sometimes referred to as connected TV or hybrid televisions. It is the first generation of such devices that are connected to the worldwide web and therefor can provide digital content in a non-linear way for the first time.

Since then the market and the amount of Smart TVs has been growing fast. With that the HbbTV standard evolved and more and more broadcasters have build an app for their broadcast streams. Worldwide more than 25 countries (mainly in Europe) have adopted the standard and broadcasters have developed around 300 apps that can be viewed by more than 43 million sold devices with HbbTV support[2]. Numbers are growing. It shows the tremendous potential of the market and the beginning of a paradigm shift from linear TV streams to non-linear media content on demand.

[1] `https://www.youtube.com/`

[2] `https://www.hbbtv.org/news-events/hbbtv-ibc-2016-services-and-devices/`

1.1 Motivation

By introducing Internet to the TV it automatically opened the door for web technologies to become standards on these devices. While early setup boxes like Chromecast[3] enabled some web-browsing experience, HbbTV was the first real technology that brought websites to the big screen. Instead of just watching a stream with linear content, HbbTV supported devices also provide web sources that contextually fit to the viewed content and allows the user to navigate through an app to experience more media offers.

These apps are web pages with JavaScript heavy functionality that are rendered in proprietary browsers. Unlike normal browsers though there is no navigation menu or status bar. The page is rendered with a transparent background so that you can show the TV stream in the back while navigating through the app. Depending on the TV manufacture the embedded browser is mostly a clone to the existing desktop browsers though with less compatibility. Early HbbTV supported devices are compatible with desktop browsers that have been shipped more than 10 years ago. Especially since HbbTV runs a lot of JavaScript to show its content in a dynamic way this makes it hard for developers to build their apps.

Compared to modern web development where almost everything is made out of web components, build together in a modular fashion using tools like React, Polymer or Angular, HbbTV apps are still built like JavaScript single page applications from the past. Not only because the technology within the browser doesn't allow using the latest web technologies also the developer integration with common used tools on the developers machine is not possible due to device boundaries. Since the internet was around way longer than the fact that it is supported on TV devices, the process of web development became an own industry and the surrounding tooling an own market. Companies constantly build frameworks, tools and integration that makes building complex web apps easier and maintainable. Especially browser vendors like Google or Mozilla are interested in providing an excellent developing experiences as this has become the main differentiator between browsers these days. After web technologies were standardized by the W3 Consortium[4] the competition between browsers is not based on the question of who can interpret the HTML code better and therefor render the page more correctly but more which browser supports the latest recommended standards and fanciest technologies.

Unfortunately this is not the case for all embedded browsers in Smart TVs. The HbbTV standard was developed as a superset of HTML and JavaScript. The specification itself recommends support for certain APIs but doesn't require the manufacture to embed the latest browser and all their features. This is partially due to the fact that not all web technologies are applicable on a television. For example there is no WebRTC support because most of the TVs don't have a camera built in. Additionally an HbbTV application was supposed to only add contextual information to the provided broadcast stream. Building complex web applications was never considered becoming reality in the first place. However as more broadcaster discovered the opportunities that HbbTV

[3] `https://www.google.com/intl/de_de/chromecast/`
[4] e.g. latest HTML specification: `https://www.w3.org/TR/html5/`

can bring to the audience more people were interested in finding new ways to deliver interactive content to anyone in front of the screen.

This yields a problem that has been around in web development for ages but almost died due to the standardization of web technology. With more support for the HbbTV standard the manufacture started to improve the functionality of embedded browser not only by adding more computing power to Smart TVs but also by providing better compatibility to already accepted web standards. In addition to that the HbbTV standard itself evolved and requires now certain technology to be supported in order to allow the TV to label itself as HbbTV compliant. The problem with this is that not everyone buys a new TV as soon as there is a better one on the market. Especially since they last longer than usual mobile phones or computers the update cycles of televisions are long. Compared to desktop browsers which update themself automatically and have reached version numbers in the fifties, the browsers within today's Smart TVs don't update and stick as long as the owner keeps the television. This creates a highly fragmented user market with tons of different devices over time that all support a different level of HTML and JavaScript. Building qualitative HbbTV apps that suppose to run on the majority of devices in peoples households becomes super time-consuming and expensive since you don't know if the device can execute your scripts or if you've used functionality that is not supported. As a developer you not only have to own all the TV devices which is literally impossible, you also need to manually test your app on each one of these. This process is cumbersome and not scaleable.

1.2 Objective

With the HbbTV standard not being older as a couple of years the development and quality assurance process for building apps for the Smart TV is lagging behind modern web development standards. Due to the high fragmentation of televisions on the market it is almost impossible to ensure 100% functionality for each individual TV. In addition to that since the browser that renders the app is embedded on a remote device it appears to be way more difficult to not only build HbbTV applications but also to debug them in case a certain TV doesn't run a certain functionality. Furthermore since HbbTV is a fairly new standard on the market it has not even closed attracted a developer community around the technology compared to the modern web on desktop and mobile. Not more than a handful frameworks have been open sourced so far that can simplify the work of an HbbTV app engineer. Most of the problems still have to be solved individually which increases the probability of introducing errors and issues that have to be fixed.

The goal of this thesis is to mitigate common developer issues when building HbbTV applications for arbitrary Smart TVs by providing a set of tools that has been proven to be useful for modern web development on desktop and mobile. The way how we build apps should not be any different when switching between computer, handheld or television devices. Due to standardization and integration work that has been done so far we gathered a reliable set of tools that work perfectly with each other and is known by every programmer. Current HbbTV app development is extremely painful when it

comes to debugging and web-authoring. Often people build their homegrown solutions which is not more or less than a panel with logging messages. There is no real interaction happening between app and app engineer. This makes fixing bugs almost a trial and error process.

By bringing modern web tooling to the TV we don't only improve developing experience but also increase the velocity and quality of shipping HbbTV apps to the market. Looking at televisions being no different from other internet connected devices these days we should try to adapt technologies for quality assurance like automated testing approaches to improve the way we test HbbTV apps. Especially since the mobile market is similar fragmented than the TV market we should take a look at how mobile solved this problem and try similar approaches to solve it. Ultimately the developer should not care about whether he is testing a desktop, mobile or HbbTV app. All he needs to know is the app itself and the features he wants to test, the rest should be identical to any other app or website he builds.

1.3 Scope

As the title of this thesis discloses I will try to design and implement a development and test automation platform for HbbTV based applications. We should look separately on these objectives as they fulfill different purposes.

The development platform should provide engineers a way to debug and interact with the app on the TV in a similar fashion than they would do on the desktop using their favorite browser. Recent browser statistics[5] clearly show that Google Chrome is not only the most used browser in the market but also the most preferred browser for web development. The main reason is its built-in Chrome Developer Tools[6] that *"give[s] developers access to the internal workings of the web browser and web apps"* [Barron, 2015]. Since it is so popular and well-known to engineers I will find and implement a way to use the Chrome DevTools application to debug HbbTV apps directly on a Smart TV. The DevTools app itself is nothing more than a usual web app that can be used by the Chrome browser. It is publicly available on GitHub[7] and can be used with respect of Googles license agreement[8]. That being said, the actual technical work here is not to rebuild the DevTools application but to reverse engineer the communication that happens between that app and the browser in order to enable information exchange and debugging commands. Due to the fact that the browser that renders an HbbTV app does not disclose these informations we need to find a way to provide these by injecting a script that can send us all the details of the HbbTV app based on the Chrome DevTools Protocol. Due to limitations in time and technical support of the browser that is running on the TV

[5] Browser statistics is hard to measure and always depends on which user base you track. However regardless if you look at more general statistic like StatCounter (`http://gs.statcounter.com/`) or a more web developer oriented one like W3Schools (`https://www.w3schools.com/browsers/`) it shows that Google Chrome has by far the biggest market share worldwide.

[6] `https://developer.chrome.com/devtools`

[7] `https://github.com/ChromeDevTools/devtools-frontend`

[8] `https://github.com/ChromeDevTools/devtools-frontend/blob/master/LICENSE`

it won't be possible to support all features that the DevTools app provides. The work concentrates only on parts that are relevant to HbbTV app developer and which are feasible with all the limitations of the browser on the TV. To be more specific, they will be able to see the DOM structure of the app as well as its sources and network traffic. Furthermore it will be possible to add attributes to DOM nodes, alter or remove them. In addition to that they will be able to use the console tab to run arbitrary JavaScript code getting executed on the TV in the same runtime environment as the app. With that the developer will be able to access the global scope of the runtime and therefor also inspect certain states of the app. There will be two different ways provided to inject the script that enables the communication with the DevTools application.

Based on this work the test automation platform will allow to run automated tests using the WebDriver protocol[9] on real TV devices at the Fraunhofer Institute. As foundation I will build a test automation driver based on Appium's core technologies. Appium[10] is a framework for mobile test automation and home for a lot of drivers that allow us to run tests on iOS, Android, Mac OS and even Windows applications. Using a Raspberry Pi[11] as a proxy I will equip TVs in the device farm of the institute to run that driver and connect it to a Selenium[12] Grid server in order to run these tests in parallel and with any programming language using any library that supports the latest WebDriver protocol. With continuous integration and delivery tools like Jenkins[13] we will hook up an actual HbbTV app that will be tested on different TVs every time someone changes the code in the repository. With the help of various test reporters we will be then able to identify issues that come up during testing to spot bugs that were introduced by the developer immediately. Since the WebDriver protocol is specialized for browser automation some of its specifications won't apply for TV devices. As the only input device for televisions so far is the remote control the test driver won't cover the whole protocol. User prompts, element interactions like click and keyboard actions are not designed to work on a TV. Also due to time limitations I will not cover screen capturing or frame switching as they are not important commands to ensure the quality of HbbTV apps.

With both platforms in place the developer will be able to debug and test his applications with modern tooling that is used in today's web development. Since the technologies for both is based on common used and well-defined standards we don't restrict our self to another homegrown solution instead we allow the integration to other tools which makes us less bound to a certain debugging or test process. As described in figure 1.1 the developer will have three different ways to utilize the TV. One way is to use a TV that was equipped with a Raspberry Pi running the automation driver on it. The Pi will act as a proxy and allows to automatically inject the automation script as well as track network data to display requests and responses made by the HbbTV application in the DevTools app. Because of the amount of edge cases and issues this proxy can run into, there will be no guarantee that this approach is bullet proof and easy to use. Running

[9] https://www.w3.org/TR/webdriver/
[10] http://appium.io/
[11] https://www.raspberrypi.org/
[12] http://www.seleniumhq.org/
[13] https://jenkins.io/

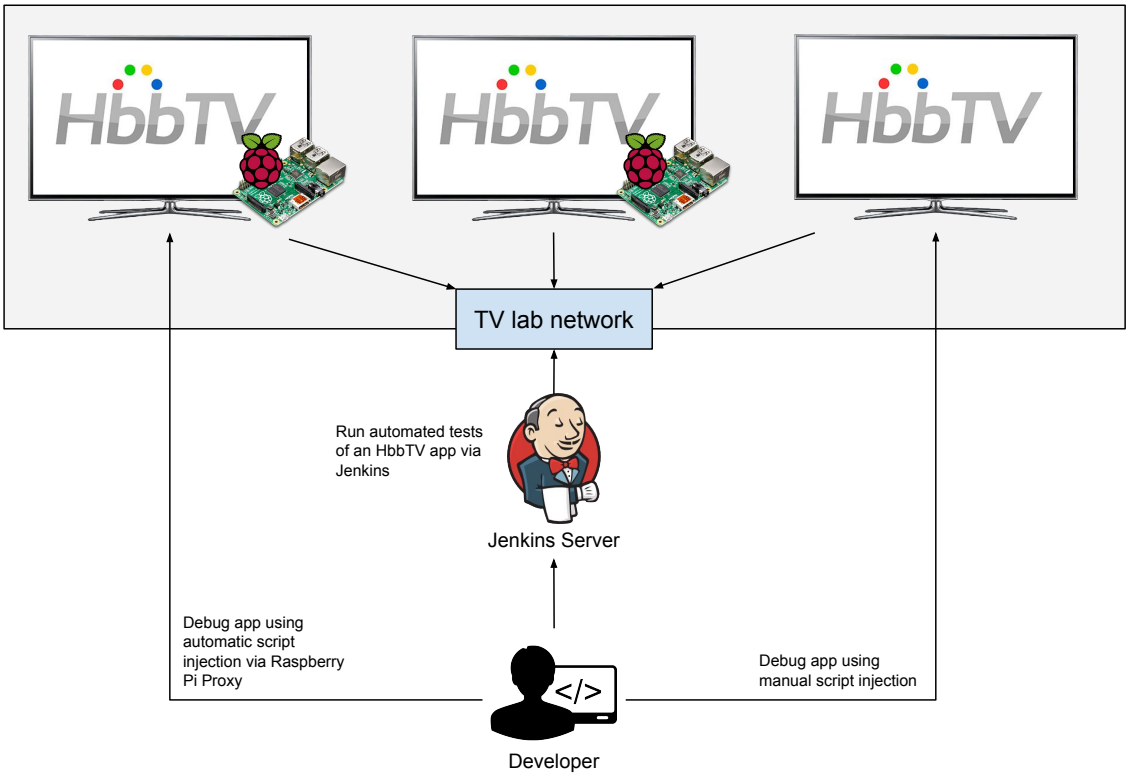

Fig. 1.1: Big picture of how developers will be able to develop and test their HbbTV application

this kind of setup requires some amount of effort to maintain it. That's why there will be another way to debug and test HbbTV apps without a Raspberry Pi. This solution will require some manual changes to the source code of the app to make it work.

1.4 Outline

Before we jump into the details of how all this will be implemented we will first look at some fundamentals of the technology that will be used. Next to the HbbTV standard, its market support and distribution as well as how developers build HbbTV apps these days, there will be a section on the WebDriver protocol and how automated testing changed the industry from moving from waterfall to an agile methodology thanks to continuous integration and delivery. In the next chapter we define the requirements and find out what is needed to solve the issues that are mentioned in the motivation section. Once we have that we can define our concept where all components of the created tool are described in detail with explanation on how they interact and work with each other. It will also outline how Appium works and how this tool fits into Appiums Ecosystem. In the implementation part of this thesis we then go into more detail on how certain parts

are build and which technology is behind it. It will explain how certain components have to be setup as well as how you need to configure all network components to create your own TV grid. At the end we conduct an evaluation of a similar product that tries to solve almost the same problem in a different way before we end up with a summary of the work, giving an outlook about future work and interesting features that could be implemented next.

2 State of the Art

There are two main standards/protocols that are used in this thesis and will be described in depth in this chapter: the HbbTV standard and the WebDriver protocol. Both are fairly new standards that have been defined within the last 5 years. Another important protocol for this work is the Chrome Remote Debugging protocol which is used by the automation driver to drive the WebDriver tests and to get data out of the HbbTV app. This thesis tries to connect these standards to allow interoperability between them to increase the level of integration to a wide variety of tools that has not been possible before.

2.1 HbbTV

Television devices started to become connected to the internet already before HbbTV was invented. With the so called Set Top Boxes or Smart TV Sticks customers were able to connect the TV to an internet capable device that provided a content stream. Big companies like Google, Amazon or Apple offered such devices. The problem with this approach was that if a developer wanted to build an app he had to create this app for each individual platform. This is a really time-consuming and not scaleable process as the way you build apps for such platforms was extremely different. After game consoles like Sony PlayStation or Microsoft's xBox became internet connected devices too there even was another way to transform the normal TV to a Smart TV. It made it even more difficult for content providers to deploy their services to all TV platforms.

With the open ETSI standard called HbbTV a new way was build to create digital applications for broadcasters and content providers that runs on every TV supporting that standard. It *"is a globally initiative technology mainly developed by industry leaders"* [Xu et al., 2016]. The standard itself introduces only a handful technical components and is mainly based on existing standards. HbbTV applications are HTML web pages with additional features that enable certain interaction between broadcast and broadband. It is sometimes referred to as Hybrid Television since the TV device receives data from both of these sources in parallel. This means that from a running broadcast stream it allows customers to open apps that provide additional content to the image on the screen. From that you can start even more applications that provide different content in non linear fashion. We have to differentiate here between two different types of apps. There are broadcast-independent applications that are not connected to any broadcast service and are downloaded and accessed via broadband. The other type are broadcast-related applications that can be opened when a certain broadcast service is used. They can start automatically or explicitly upon user request. The advantage of HbbTV here

is the independence to the used device. Content providers can build and deploy apps which can be used by any device if the standard is supported.

If a consumer is equipped with an HbbTV supported Smart TV he will usually see a small image somewhere on the screen that offers to press the red button on the remote control. Pressing it triggers an event within the app that opens the HbbTV application. Once the user switches the channel the same happens again with a different app. Depending on the broadcaster each broadcast signal contains a small AIT package that contains main information like application ID, version, autostart state and application URL of the HbbTV app. Figure 2.1 demonstrates the workflow of this process.

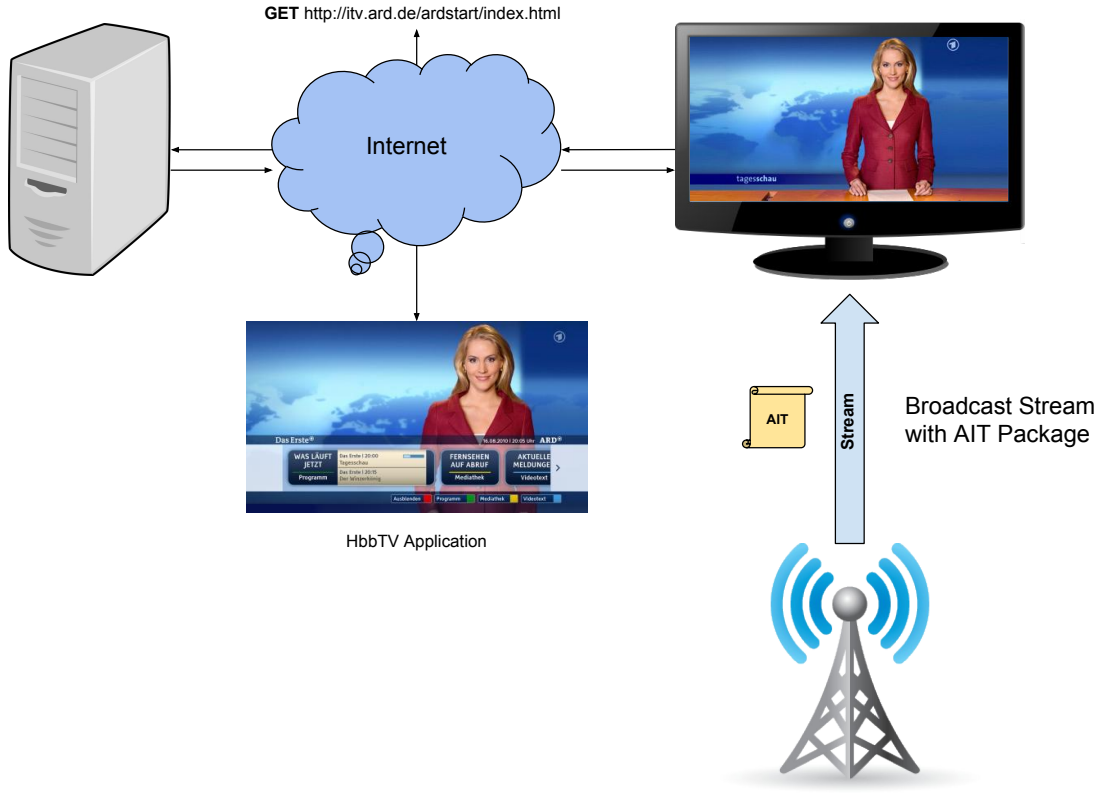

Fig. 2.1: App delivery workflow of HbbTV applications

Depending on the autostart information within the AIT package the app either starts automatically as soon as the broadcast stream starts or only when the user pushes the red button. Most HbbTV applications these days use the autostart feature to display a small indicaticator image to show that an HbbTV app is available to use. Once the app opens, it either takes up only some part of the screen or fulfills the whole display. In general though the broadcast stream is never getting disrupted as the application only acts as an additional service to it.

After the first specification was released in 2010 by the *"HbbTV forum and published by ETSI in the specification TS 102 796"* [Podhradsky, 2013] initiators reached out to CE manufactures to get it implemented in the next generation of TVs. Since then the number of Smart TVs with HbbTV support has increased drastically with more than 43 million devices sold and over 300 apps deployed in 25 countries[1] [2]. Alone in Germany there are over 120 HbbTV supported channels[3] generating roughly 320 million hits monthly[4].

Almost two years later a new specification was released (HbbTV 1.5) that added support for MPEG DASH and MPEG CENC. MPEG DASH is an ISO standard for adaptive streaming of IP videos. Depending on the users bandwidth and CPU processing power it ensures continuous playback and helps to improve the stream experience in general by monitoring the CPU utilization and/or buffer status. If e.g. for any reason the user looses bandwidth the player is able to lower the quality of the video so that it still can be played without disruption. Due to support issues not many HbbTV apps use DASH these days. Most videos are still loaded in a progressive way. The MPEG CENC on the other side helps with common encryption for ISO based media format files and MPEG-2 transport streams. As a content provider using this feature you only have to encrypt your video once. The user then has to decrypt it by getting the key from one of many DRM systems.

The HbbTV 1.5 standard is supported by the majority of devices these days, however many CE manufactures already are about to support the latest HbbTV spec version 2 with their new devices on the market. It introduces a lot of new features and supports more web technology standards. While previous HbbTV apps only work with a version of CE-HTML which is an XHTML-based standard specifically for consumer electronic devices on UPnP networks, the latest specification supports HTML5, CSS3 and Web Sockets. Another new feature is the ability for synchronization of audio, video and data streams that enables the consumer to watch a certain broadcast stream while listening on the audio via broadband. This opens a wide variety of use cases like services for simultaneous translations of videos. It can be combined with the new introduced Second Screen Integration for smartphones and tablets which allows starting broadcast streams from your personal handheld on the TV. Due to its duplex communication ability it allows to also start videos from your HbbTV app on your mobile device. This works with multiple devices and also provides a lot of opportunities for e.g. gaming applications where phones can be used as remote controls. In the video area the specifications will support new streaming formats like HEVC, also known as H.265 and MPEG-H Part 2, DASH for DVB or push-on-demand where users are able to download certain videos on a local storage to watch them later if desired.

It is worth mentioning that version 2 of the spec was never really released by ETSI. It got surpassed by version 2.0.1 which fixed unclear specifications and removed some errors. It also shipped some additional specifications that were required to adapt the standard

[1] Source: `https://www.hbbtv.org/news-events/hbbtv-ibc-2016-services-and-devices/`
[2] A complete list of all countries can be found in Listing 1 in the annex section
[3] Source: `https://trifinite.org/hbbtv/trifinite_hbbtv_channel_list.html`
[4] Source: `https://goo.gl/eg8Sfs`

in Italy and United Kingdom who have decided to move from a similar standard called MHP to HbbTV. These specifications define additional caching rules for HTTP/1.1, more details on higher display resolution and other technologies that were defined in MHP.

An additional standard to HbbTV was released by ETSI shortly after HbbTV version 2. This standard is called Application Discovery over Broadband and is not part of the actual HbbTV specification. However it adds an important addition to HbbTV that is used if cable service providers block the AIT of the broadcast stream. These providers have a different idea of interactive TV services and usually want to promote their own platform. Once the AIT package is blocked the HbbTV application can't be loaded by the TV anymore since information about application ID and URL are missing. Application Discovery over Broadband is a workaround to solve this problem.

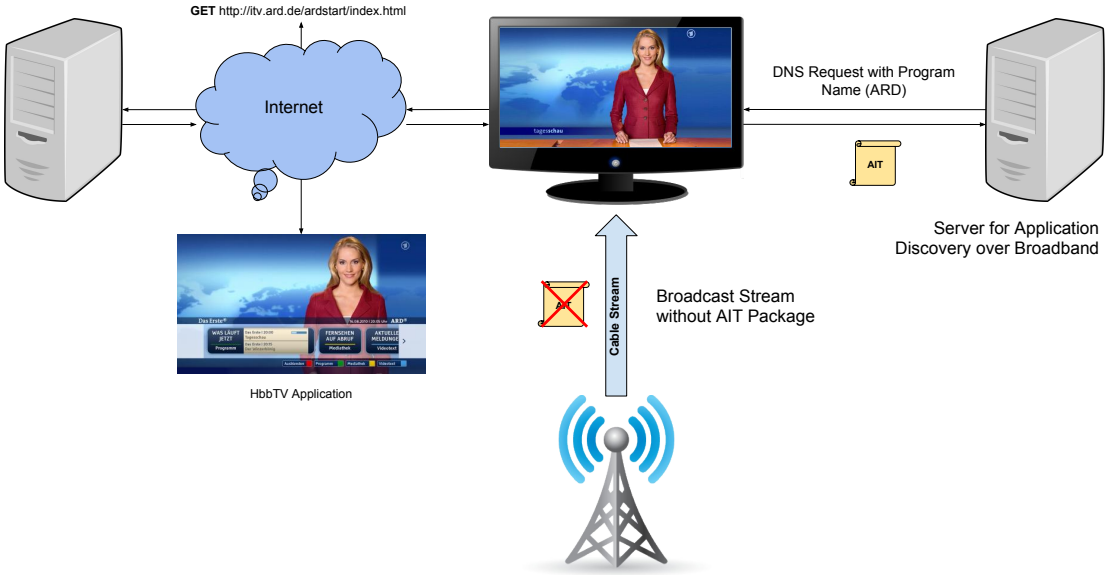

Fig. 2.2: App delivery workflow of HbbTV applications over broadband discovery

As described in figure 2.2 the TV gets the AIT package from a dedicated server based on the program name. Each broadcaster usually has their own DNS root server and an own AIT server. However there is one main HbbTV DNS root server which routes the DNS request from the TV to the server of the broadcaster.

Pay, cable, IPTV or Sat-Platform operators as mentioned above sometimes provide instead of HbbTV their own service apps with a custom GUI. They usually require to use some sort of additional hardware which don't support HbbTV. Even though the TV is connected to the internet and supports the standard it is still not possible to open up an HbbTV application. As solution for this problem the consortium around the standard invented Operator Apps which allows connecting operator and HbbTV

applications on one Smart TV. The idea is that the user can switch between both worlds as a functionality within the TV. Anyone who has a contract with the TV manufacture can become an operator. Each operator app has to authenticate to the TV to avoid abuse. Unlike HbbTV applications an operator app can run in the background at all times and can display messages on the screen as well as overwrite key functions like *"P+"* or *"P-"*. They usually open by pressing *"EPG"* or *"Menu"*. In the specification of operator apps which is part of HbbTV version 2 the consortium emphasized that operator and HbbTV apps can live seamlessly together.

Looking back at the standard that was first specified 7 ago HbbTV has taken a very interesting development. From being able to show more than a digital and interactive teletext to supporting HTML5 and Second Screen it became quite comprehensive. This enables broadcasters to use this service in a variety of ways.

2.1.1 Example Applications And Use Cases

All these features allow broadcasters to implement not only an additional content outlet but also to create tailored advertisement strategies to a specified audience with methods like geo targeting or targeted advertisement. Geotargeting can be used in HbbTV applications by leveraging a users IP address in order show regional products or services. Combined with a broadcast stream it enables interesting opportunities. In an advertisement campaign launched by Germans private broadcaster RTL they promoted a pharmaceutical product with help of an HbbTV app. The pharmaceutical *"Wick Medinait"* was advertised along with a banner that showed the regional weather forecast. It had the effect that the HbbTV app banner supported the TV spot so that it increased the impact of the advertisement itself.

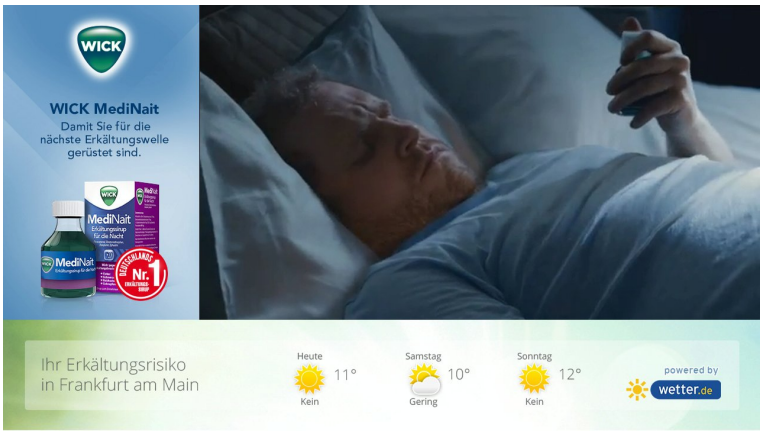

Fig. 2.3: Use case of geo targeting via HbbTV

Image Source: https://goo.gl/rab8XD

This is also called as Addressable TV advertising. It *"enable[s] advertisers to selectively segment TV audiences and serve different ads or ad pods (groups of ads) within a common program or navigation screen. Segmentation can occur at geographic,*

demographic, behavioral and (in some cases) self-selected individual household levels [...]"[Gartnet, 2017].

Another interesting use case outside of advertisement is content authoring of HbbTV applications. Usually an HbbTV application is a custom web app that is build to serve a specific content for a service or show. Once you want to promote a new service or show it requires to build a new app with new content. This takes time and costs money. A solution to this problem was adapted from the web. By using a Content Management System (CMS) like Wordpress[5] broadcasters can create or modify the content of their apps using a simple web interface.

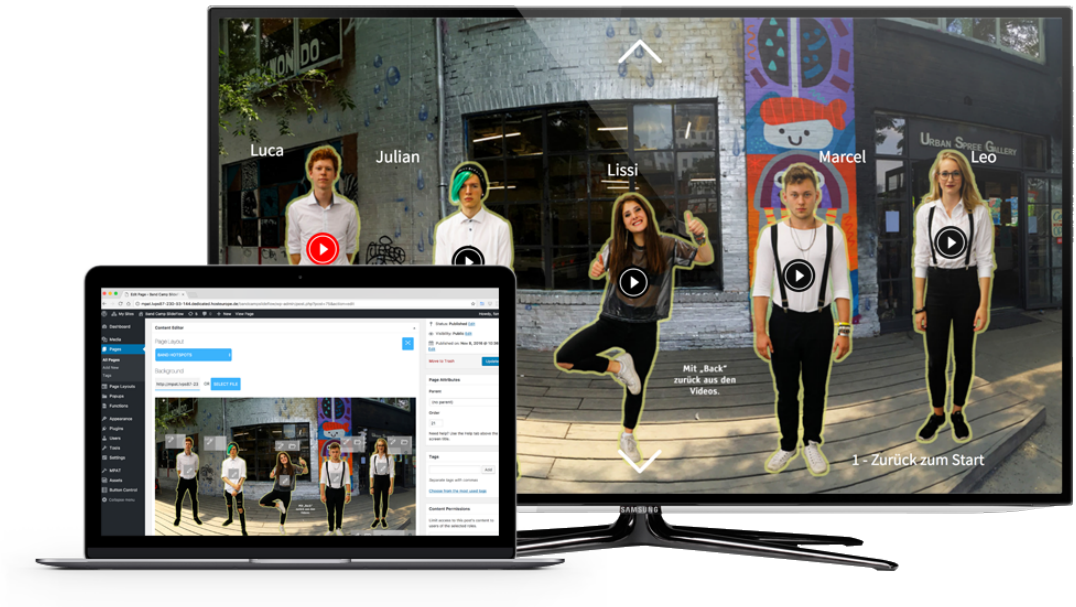

Fig. 2.4: Modifying HbbTV content via a web interface

Figure 2.4 shows how this looks like with a tool called MPAT[6], a project that was funded by the European Union and developed and led by Fraunhofer Fokus. It uses Wordpress as CMS to enable authors to modify content, look or functionality of an HbbTV app. Thanks to its plug&play functionality the tool can easily add new features like Chat and Video or Image Galleries without requiring someone to implement it. With that it reduces the risk of unexpected behavior as these plugins are already tested against common Smart TV devices.

[5] `https://wordpress.org/`
[6] `https://www.fokus.fraunhofer.de/en/fame/project/mpat`

2.1.2 HbbTV Runtime Environment

The Runtime Environment of an HbbTV application is formed by the browser and an Application Manager who *"evaluates the AIT to control the lifecycle for an interactive application"*[ETSI, 2012]. The browser on the other side, which is in many cases a clone of an already existing web browser with a subset of supported features, is responsible to render the HbbTV app. The TV receives an AIT package, an A/V signal as well as application data and stream events from the broadcast stream. While the A/V stream is processed like a standard non-hybrid DVB stream it still can provide some information like channel list or tuning functions to the runtime environment. Furthermore receives the runtime environment the application data and event streams from the object carousel using a DSM-CC client which is a toolkit defined in the MPEG-2 standard that provides a control channel to access the broadcast stream and push data to the Application Manager.

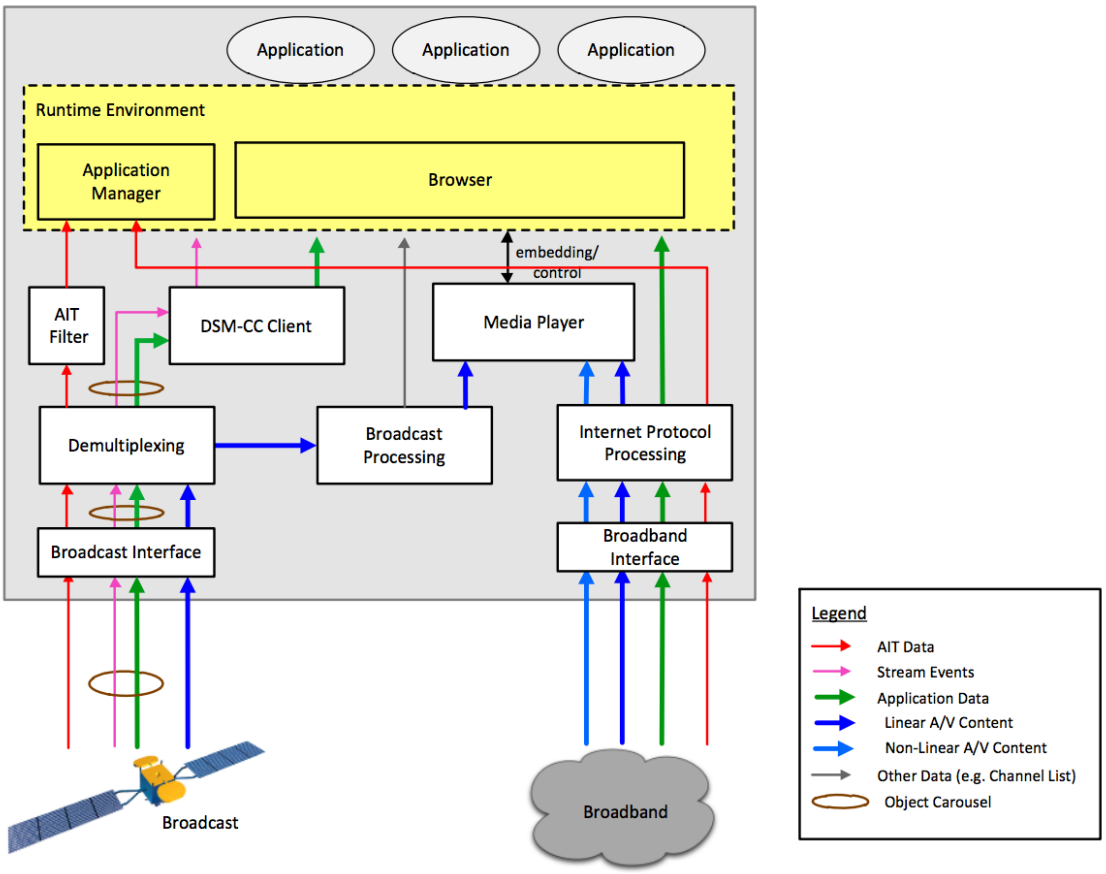

Fig. 2.5: Functional components of a hybrid terminal

In order to allow the HbbTV app to embed the running broadcast stream within the application as a video, the Media Player component is required which receives that

data after the broadcast stream was processed. It can also receive A/V content from the broadband source in linear or non linear form.

As described in section 2.1 once the HbbTV application is loaded via broadband stream, it starts either directly by the end-user by pressing a dedicated button on the remote control, in response to an event stream or by an already running application, e.g. when the user wants to open an application within the application. With HbbTV version 2 it became also possible to start the app with a companion screen. However it is often the case that the app is a broadcast related autostart application which displays a *"Red Button"* notification to inform the user that an application is available. These are called *"Red Button"* applications. To not disrupt the video stream they only show something like a small image and leave the rest of the screen transparent using CSS. This has been agreed in a guideline to not annoy the viewer with undesired overlays and to allow a uniform experience for the application start. If no action is taken by the user this indicator disappears again. Since radio services don't have this indicator, because they don't use the video channel, the full user interface will be displayed. Some applications like digital teletext apps are not run with autostart enabled. They have to be specifically triggered by the user without notification, in this case using the teletext button on the remote. Triggered one time it starts the digital teletext which is an HbbTV based application. If triggered twice it closes this app and starts the standard old-fashioned teletext. By pushing the button for the third time it closes also that.

HbbTV apps are not only tied to a specific broadcaster. The so called broadcast-independent applications *"can be electronic programme guides (EPGs) or "TV editions" of existing web services such as flickr, YouTube and very many more that may be provided by the big brands as well as on a regional level or even by individuals."*[Merkel, 2010]

2.1.3 Development of HbbTV Applications

As mentioned in 2.1 HbbTV applications are written in CE-HTML which is an XHTML based standard to build web pages for consumer electronics like televisions. With the release of HbbTV version 2 support for HTML5 was added which allows developers to use the latest web technologies in their apps. Until the majority of devices have support for that, apps will be still delivered in the CE-HTML format. Per specification the doctype of the document needs to be either XHTML 1.0 strict or has to have an HbbTV specific declaration which is shown in listing 1.

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE html PUBLIC "-//HbbTV//1.1.1//EN"
3   "http://www.hbbtv.org/dtd/HbbTV-1.1.1.dtd"
4 >
5 <html xmlns="http://www.w3.org/1999/xhtml">
```

Listing 1: Beginning of an HbbTV document

At the beginning the document needs to contain an XML declaration as well as a namespace definition in the HTML tag. This is required so that the device can properly interpret the document as CE-HTML. In addition to that the page needs to be served with an *"application/vnd.hbbtv.xml+xhtml"* content type otherwise the app is going to be ignored by the television. The browser window has a fixed size of 1280x720 pixel and therefor fits perfectly in full screen on a 16:9 ratio. With embedded objects the developer can access APIs that are defined by the Open IPTV Forum in the HbbTV spec to receive TV capabilities and configuration as well as access the application manager. It extracts information out of the AIT packages such as lifecycle of the app or the autostart flag. APIs are available for configuration and settings, download manager and content download as well as parental access control and scheduled recordings. However most HbbTV apps these days only need the following embedded objects:

```
<object type="application/oipfApplicationManager" id="oipfAppMan" />
<object type="application/oipfConfiguration" id="oipfConfig" />
```

Listing 2: Embedded Objects used to access HbbTV APIs

Using the DOM node id developers can query the element and access the API. This usally happens on the *"onLoad"* event that gets triggered onced all DOM elements and JavaScript files are loaded. The event callback then executes some sort of initialization method that loads the applications and shows it on the screen.

```
var oipf = document.getElementById('oipfAppMan');
var app = oipf.getOwnerApplication(document);
app.show();
```

Listing 3: HbbTV App initialization

The only input device for HbbTV web apps is the remote control. Within the initialization process an event listener is registered on the window global object that gets triggered on keydown events. Depending on the event key code the app executes different actions and allows navigating through the app.

Since uploading changes to a production server is inefficient and time-consuming people have developed some tools to improve the development environments of HbbTV app developers. There is a Firefox plugin that emulates an CE-HTML-like environment in the browser called FireHbbTV[7]. It simulates the TV Remote control using the standard keyboard, displays the safe-area margin as well as supports most of the HbbTV specific APIs. Another popular tool is the Opera TV Emulator[8], a virtual machine that emula-

[7] https://addons.mozilla.org/en-US/firefox/addon/firehbbtv/
[8] http://www.operasoftware.com/products/tv/tv-developer-tools

tes an Opera SmartTV environment. With that you can have your app being served on your local machine and easily access it using these tools. However even though these are helpful tools *"they don't offer the full capacities and facilities that a certified HbbTV device will provide."* [Developer, 2015]. Therefor many developers tend to build their apps directly on the TV with some sort of debug overlay that prints custom debug messages.

2.1.4 Available test solutions

Most of the standard static code analysis tools like JavaScript or CSS linters also apply for HbbTV app development. These tools ensure that the code can be parsed by the TV device. Specifically for HbbTV apps the *"Institut für Rundfunktechnik GmbH"* authored a validator[9] that checks for common pitfalls. It not only warns you if you serve the app with a wrong content-type it also makes sure that other specifications are met, like having the right doctype or required XML namespace attributes set. However these helpers can only detect obvious issues that are relevant to all HbbTV devices. Problems that only appear on TVs from certain manufactures can not be found with these kind of tools. It usually requires manual work which ends up being very tedious and expensive. There are some services that tried to solve this problem.

A Czech company called Suitest has developed a system to run e2e tests based on a web GUI on arbitrary devices either in a remote office or in a local environment. It allows to run these tests in parallel and provides a comprehensive report about the test result over time. The GUI enables their user to simply click together one or multiple test scenarios with common actions like: open the app via URL, press a button on the remote and assert that a certain element contains a certain property. Every aspect of your test can be separated into sub steps and reused in other tests. The main idea is that you don't need any coding skills to put together a test. Figure 2.6 shows how simple the GUI looks like and how tests are being put together.

To gain access to the device the company provides a setup box that comes with infrared blasters in order to simulate remote control events. You can also inject a script into your HbbTV app to instrument either your browser or TV device to allow Suitest to run your tests. In case you don't own a targeted device Suitest provides a handful of devices from their own datacenter. It requires a paid plan though which starts with 156€ up to 899€[10].

Another company providing an e2e test solution for HbbTV apps on Smart TVs is *Eurofins Digital Testing*[11]. Next to HbbTV applications they also offer multiple device types such as Set-Top Boxes, standard web-apps or mobile apps on tablets and phones. Their TestWizard Manager[12] controls a set of TestWizard Robots that contain a different variety of devices depending on the requirements of the application. Unlike Suitest the company offers a development tool called *ScriptStudio* that allows customer to also write automated tests *"without requirement for extensive knowledge of scripting languages"*

[9] `http://hbbtv-live.irt.de/validator`
[10] `https://suite.st/pricing.html`
[11] `http://www.eurofins-digitaltesting.com/`
[12] `http://www.eurofins-digitaltesting.com/testwizard/infrastructure/`

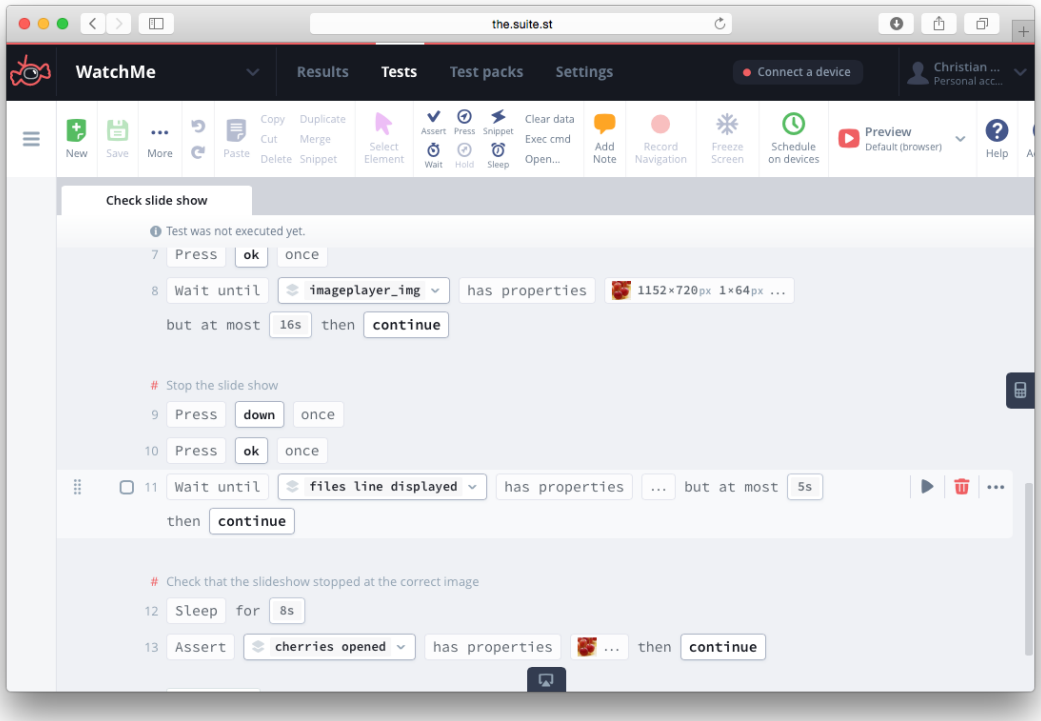

Fig. 2.6: Suitest Test Editor GUI

[eurofins, 2017]. Since other devices types that they offer for mobile and desktop can be tested via test automation scripts it is also possible to write e2e tests via Selenium and Appium. The company itself is with 22,000 employees way bigger than Suitest. They have a network of more than 200 laboratories and offer complete testing solutions not only for HbbTV but also for other key technologies and standards such as DVB or DASH.

2.1.5 Web Platform Tests

Within the last decade many organizations including the W3C tried to build standards around technologies that we use everyday. Even though these specifications were adapted by developers and companies there was still a lack of conformance, quality of implementation, performance and interoperability. In 2013 the W3C launched *"an unprecedented effort to scale up its test offering"* [Langel, 2017] in which they tried to improve the test infrastructure and documentation around all web standards that are defined by them. By streamlining the contribution and reviewing processes they simplified the way how everyone can be involved in defining and testing a standard. They published their tests and made them accessible for everyone so that anybody was able to get a clear picture of which web features are supported in whatever application they are testing. With that

every browser vendor was able to make sure that its browser is working according to the spec. They became the major contributor for writing and maintaining the tests. However the W3C also encouraged every developer to be involved in this endeavor by starting a movement called *"Test the Web Forward"*[13]. This was one of major reasons why standards like HTML5 and other technologies that are part of the HbbTV specification got reliable implemented in many browsers and platforms.

The HbbTV specification which was defined within the European Telecommunications Standards Institute did something similar to ensure that TV manufactures only sell devices that are complaint to the spec. This is important as *"lean-back consumers have zero patience meaning any malfunction in the chain will significantly reduce engagement and impact and may even result in viewers switching to another service"* [Campbell, 2015b]. In case of an update in the standard the association has to make sure that older applications are interoperable with forthcoming receivers and can coexist with them especially since devices are not easily updated by the consumer. Therefor they defined a receiver specification and a device and application conformance that sets certain guidelines and restriction to manufactures and developers to fulfill the standard and enforce on a common sense. Similar to the W3C a test suite[14] is provided to manufactures *"to certify their own devices or have their devices certified by an HbbTV Registered Test Centre."* [Campbell, 2015b]. Once they are certified they can use the official HbbTV logo for their device products as proof for compliance to the spec.

2.2 Test Automation

There are a lot of different ways on how to test software. Depending on its type and the scope of the tested functionality there are multiple testing methods that can be applied. From unit testing, that only covers individual software components and code segments, to load testing, which makes sure that a specific performance metric is met, these methods are used at different stages of the development lifecycle. With test automation these testing methods can be executed with use of special software that controls the execution of tests and compares the actual state with a certain predicted outcome. This happens in an automated fashion so that the test itself is mostly triggered by a certain event (e.g. code changes) in a CI/CD like environment.

From the agile world where coding and testing are one process a concept developed by Mike Cohn became popular that describes how *"an effective test automation strategy calls for automating tests at three levels: unit, service and UI"* [Cohn, 2009b]. It explains that *"unit testing should be the foundation of a solid test automation strategy and as such represents the largest part of the pyramid"* [Cohn, 2009a]. On the other side user interface tests which are placed at the top of the pyramid should be as small as possible as they are brittle, error prone and time-consuming. However they simulate real user scenarios and ensure that the functionality is working on all levels, from database over the backend to the user interface. According to Googles philosophy *"focus on the user and all else*

[13] `http://testthewebforward.org/`
[14] `https://www.hbbtv.org/resource-library/#testing-information-and-support`

will follow"[Google, 2017c] you could convince someone that writing only e2e tests is a good idea even though it would object the test automation pyramid. Many developers follow this bad practice of having to many e2e tests (inverted pyramid or ice cream cone pattern) or a lot of unit and e2e tests but almost none integration/service level tests (hourglass pattern). A good rule of thumb here is splitting tests into 70% unit tests, 20% integration tests and 10% e2e tests. Especially when working as an HbbTV app developer the desire to test apps directly on multiple TVs is high due to the high device fragmentation in the market.

Testing software from end to end is a hard problem to solve. It either requires manual work, which costs a lot of time, or engineering effort to automate this process. Over the last years the tech industry moved from manual QA to automated testing due to the *"shift from Waterfall to Agile development"*[CrossBrowserTesting, 2017]. Software these days gets released in shorter cycles and feedback loops. Especially due to the growing popularity of DevOps, development and operations teams are no longer soiled and engineering work takes up the entire application lifecycle, from building the app to test and deployment. This allows not only to innovate for customers faster and adapt better to changes in market, it also forces you to increase the frequency and pace of releases in order to improve the product faster overall. Software has become *"an integral component of every part of the business"* [BEPEC, 2017]. One fundamental practice in DevOps is Continuous Integration and Delivery. It improves the software quality by running automated tests on regular bases after code changes were committed. Once tests have passed, developers can automatically push their code to production with a high level of confidence that no regressions were introduced. This can only be accomplished with proper tooling. In order to test software from end to end a framework called Selenium has become the tool of choice for many developers.

2.2.1 Selenium

Selenium is a set of tools and libraries to automate mainly web browsers and mobile devices. It can interact with them to simulate user actions like click or inputs and assert certain states of web applications. These libraries are open source, available in almost any code language and are used to run e2e tests. It is supported by all the major browsers where each one provides some sort of automation driver that speaks the Selenium/WebDriver protocol. The communication is based on HTTP requests between an automation server and the client. Each test automation is coupled to a session id. Once a session is initiated you can communicate with the automation server via rest API. Such a server can be either a Selenium server or directly a browser automation driver. A Selenium server can be used to setup a grid of automation drivers to load balance requests based on its capabilities to a specific browser in the network. These browsers can be installed on different operating systems with different environments. It manages all initiated sessions and reroutes them accordingly. A browser automation driver can register itself as a single node to this hub server to make itself available for others to use. It allows to speed up the execution time by running tests on multiple browsers at the same time.

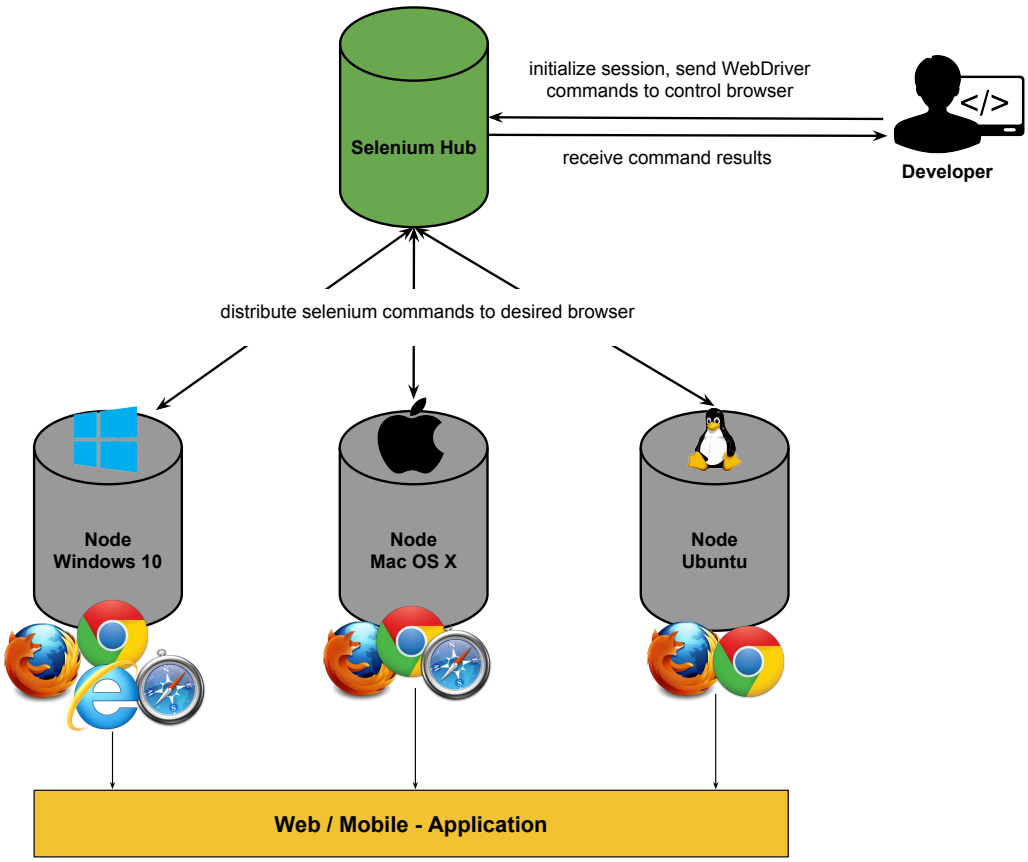

Fig. 2.7: Concept Flow of a Selenium Grid

It is important to point out the difference between Selenium and WebDriver here. Selenium was developed in 2004 by Jason Huggins, an engineer at Thoughtworks[15] who tried to build a tool to test web applications that require frequent testing. The tool was called *JavaScript Test Runner*. It was able to interact with the web page in order to call certain actions on it. Due to the Same Origin Policy that was introduced by browser vendors to prevent JavaScript from accessing data from pages that are not within the same domain, Huggins and his team had to build a proxy server to overcome this. It injected a script to the web page to talk to the proxy server and therefor enable the interaction with the browser environment. This proxy server was released to the public as Selenium Remote Control and is the first version of Selenium. In 2007 Simon Stewart who was also working at Thoughtworks build another tool to run browser tests which he called WebDriver. His tool was not a proxy. It communicated with the browser directly to use its own engine for automation. In 2011 both tools got merged into one project that was called Selenium WebDriver (or Selenium 2.0). Over years it became the number one choice of developers for browser automation so that originators and the community

[15] `https://www.thoughtworks.com`

around the project decided to make the Selenium protocol (also called JsonWireProtocol) a W3C standard. In 2016 this standard was released by the consortium as candidate recommendation after it was fully implemented in two major browsers (Firefox and Internet Explorer). It was officially called WebDriver specification. Even though Selenium and WebDriver are referencing the same project and technology it still differs in some minor specification details. The Selenium project itself nowadays only contains the tools that can be used to run tests with WebDriver. Every part that was responsible to automate the browser became obsolete after all browser vendors stepped up to implement the automation engine directly into the browser.

2.2.2 Appium

After the success of Selenium and the growing market of mobile devices and applications the demand for automating mobile phones increased more and more. Even though the big player in the market (Apple and Android) did provide tooling to create automated tests it still was time-consuming and cumbersome as both platforms forced developers to use their own platform specific technologies. With that it not only took twice as long to build the app since they had to be written in Java for Android and Objective-C for iOS respectively, they also had to be tested with two different frameworks. Appium aims to solve this problem by providing a cross platform solution for native and hybrid apps. It allows developers to write automated e2e tests on both platforms using the WebDriver protocol. It extends that protocol to introduce mobile specific actions like swipe or touch gestures to the developer. Moreover Appium supports hybrid applications as well which are mobile web apps that are wrapped within a tiny native container so that you can write one single app and deploy it for both platforms. With that it can also run a random web page on a browser on the mobile device. Since it leverages the native test automation frameworks it doesn't matter if you run your test on an emulator or simulator on your computer or a real device that is setup in a test lab. Since the WebDriver protocol is universal it doesn't require writing tests for Appium in a certain language. Similar to Selenium it is possible to write tests with any tool that supports the protocol. It also allows connecting to a Selenium Grid to not only have a browser but also a mobile grid available to use for automated tests.

2.3 Chrome Remote Debugging Protocol

As described in section 2.2.1 when Selenium merged with WebDriver they started to communicate with the browser directly in order to interact with the page. When the first automation driver for Google Chrome was released, it used an internal protocol called Chrome DevTools Protocol to instrument and debug the web page. By that time this protocol was mostly utilized to run the Chrome DevTools which is a developer tool to inspect the wepage and profile the underlying rendering engine called Chromium. The protocol is divided into a number of domains where each *"domain defines a number of commands it supports and events it generates"* [Google, 2017a]. If the user activates

a flag[16] when starting the browser, a TCP server is initiated and the API can get accessed via a WebSocket connection. This allows to remotely control the browser in almost any possible way. The protocol itself is comprehensive and allows besides receiving information about the state of the page a lot of debugging and profiling tools. Since it is part of WebKit which is the layout engine for many browsers - including Chrome (which uses a fork of WebKit called Blink), Safari or Opera - it gets utilized for debugging purposes in many web environments. Due to the number of already existing integrations (including Chrome DevTools) it almost makes it an universal protocol for automating a browser.

[16] e.g. in Windows: chrome.exe −−remote-debugging-port=9222

3 Requirements

The development of HbbTV applications these days is cumbersome and difficult to develop and test. Even though tools like FireHbbTV[1] seem to have found developers' interest and attraction the workflow is still not even close to current industry standards in web development. Tools like Selenium and Appium have proven that there is an high demand for developers being able to develop and test their software in a real life environment.

With the shift to a more agile software development in the industry it also proofs the demand to run automated tests in an iterative and quick development cycle. This can only happen with proper tooling. Software development in browsers or on mobile devices already successful entered that space of rapid development and quality assurance using CI/CD. A tool that tries to bring HbbTV app development to the same point has to build on this strategy. Since HbbTV apps are not much different to normal webapps in their core technology it would be confusing and unhelpful if continuous integration and continuous delivery would be a different process here. Therefor the tool has to be based on existing standards around automated testing and development. It should not only support a seamless integration to already existing tools but also allow developers to keep their current common practices in building apps similar if not equal to other areas of web development.

The referred tool of this thesis, as outlined in chapter 1.3 and more detailed described in the concept section, will be a piece of software separated into two components. Both have individually different requirements. The main goal of the first component called *DevTools Backend* is to provide developers an integration to the Chrome DevTools so that they can inspect and debug their HbbTV applications. The other component called *Appium HbbTV Driver* will be based on that component and should enable to run automated tests based on the WebDriver protocol for arbitrary HbbTV applications on arbitrary SmartTVs. Both components should be easy to use and integrate into a common and familiar development setup of web engineers.

3.1 Functional Requirements

3.1.1 DevTools Backend

The DevTools Backend component should help developers to understand the state of their HbbTV apps and give them instruments to inspect and debug them. As the name already discloses the component should do that by providing a first class support for the Chrome DevTools. This means that it serves that app on a specific port including an

[1] `https://addons.mozilla.org/en-US/firefox/addon/firehbbtv/`

overview page that lists all inspectable apps as well as an API endpoint to consume that information as JSON for 3rd party tools. To allow seamless updates the app should be a direct dependency to the actual on NPM[2] released *chrome-devtools-frontend*[3] package. It is separated into two subcomponents. An instrumentation script that has to be placed within the target environment as well as a backend server that connects to the instrumentation script to integrate with 3rd party clients like the Chrome DevTools application.

Like debugging a website in a browser it should be possible to inspect an HbbTV app on a television live using the app. Because the time of this thesis is limited and the DevTools application provides a huge amount of tools and functionality it is not required to support every aspect of the tool. Since it is technical impossible to do so it will be sufficient to only enable basic instruments that are important for the development of an HbbTV app. These are among others the *Elements* panel where the developer can inspect the DOM tree, modify DOM node attributes, create and edit DOM nodes as well as audit and modify their CSS properties. Another important part to support within DevTools is the *Console* tab. It has to provide a JavaScript console that executes the code in the same runtime environment as the HbbTV application. It has to support type suggestions and should make the results discoverable. In addition to that it is supposed to display all errors and logs that were triggered by the app including those ones that happen before page load. The *Source* tab has to give developers and insight about which files are used by the document. This includes the HTML file and all JavaScript and CSS assets as well as images. The last supported functionality from the DevTools app is the *Network* tab. It provides information on all network packages that have been loaded for the HbbTV app including their package headers and response. A basic timeline support will then give also an idea on the order in which the files were loaded. However all the data can only be captured when the component is used as proxy.

To capture network data the component should be able to act as an HTTP proxy. Used in this mode it not only captures all network traffic but also modifies the HbbTV application automatically so that it instruments the page without having to manually modify it. All HTML pages or files with an HbbTV content type have to be returned by the proxy with already injected code. However using the component as proxy is sometimes not feasible due to technical limitation therefor it should serve a script that has to be manually placed into the HTML code by the developer to unlock the same functionality without the network aspect. This launcher script has to be able to register to the DevTools Backend and instrument the page the same way as it would have been included by the proxy.

Since the script can't keep up the connection when switching pages the DevTools Backend needs to be able to seamlessly reconnect to the HbbTV application in a way that the DevTools frontend application doesn't alert the user that the connection broke. In fact it should automatically update the *Elements* tab with the DOM tree of the new page as well as change the context of the script execution to the new runtime

[2] https://www.npmjs.com

[3] `https://www.npmjs.com/package/chrome-devtools-frontend`

environment.

The page instrumentation should work for all web platforms including but not limited to Smart TV devices with support for HbbTV version 1.0 upwards. It has to run as a standalone script in a way that it doesn't interfere with any other scripts that are used on the page. Similar to the Black-box testing model, where a functionality is tested without peering into its internal workings, it should not have any expectations or dependencies nor do any severe modifications on the environment it is running in. The instrumentation script has to get all instructions from the DevTools Backend and returns the output back to it. All changes to the environment like DOM nodes or page properties have to be either instructed by the Chrome DevTools application or invisible so that it doesn't break any functionality of 3rd party scripts. It cases where a certain JavaScript API is not supported it should fail silently so that it doesn't affect other parts of the instrumentation functionalities.

Since this component is used to automate the HbbTV app using the Appium HbbTV Driver it has to introduce additional functionality to cover the requirements of the WebDriver protocol. That includes the serialization of DOM nodes so that they can be referenced via WebDriver.

3.1.2 Appium HbbTV Driver

The Appium HbbTV Driver will provide an API interface to run automated tests for HbbTV applications. The interface should be based on the latest WebDriver protocol[4] (candidate recommendation) published at the 30th of March 2017. Since this protocol is actually targeted for web applications within the browser there are numerous parts that don't apply for the TV environment. With that the driver doesn't need to have support for proxy capabilities, command contexts and iframes as these are not applicable for HbbTV environments. In addition to that a support for some parts of the specified element interaction are obsolete too. So should the driver be able to simulate key events within the app to allow navigation, it however is not necessary to support click events or to be able to clear input fields. The only input device for HbbTVs is the remote control which only has keys. Even companion screens won't provide that functionality which is why there is no reason to support it. Same goes for Actions which defines the emulation of pointer devices in web applications. This, as well as user prompts, will be ignored due to lack of compatibility. Capturing screenshots however could be supported but will be also ignored due to lack of time.

Like all other Appium drivers the HbbTV driver should seamlessly integrate into the already existing Selenium ecosystem. That means it has to be able to integrate into an existing Selenium Grid so that multiple TV devices can be instrumented at the same time. To simplify that process the driver should provide a settings page where the user can input host and port of the grid within the network. That allows an automatic registration to the grid. Furthermore this process has to consider the properties of the TV so that it is possible to define the model name or the supported HbbTV version

[4] `https://www.w3.org/TR/webdriver/`

within the capabilities of the WebDriver tests. If for example the developer requires to run tests only on devices that support a certain HbbTV version the Selenium Grid should be able to only connect to HbbTV drivers that are attached to such Smart TVs.

Similar to other mobile drivers like the iOS[5] or Android[6] driver it should be possible to define the URL of the HbbTV app that is supposed to get tested within the capabilities of the Appium session. Therefor it should not be required to setup or modify the broadcast interface to test a specific application. As long as the TV receives a broadcast stream with a proper AIT package the driver should be able to properly open up the right HbbTV application. Because of time limitations the driver doesn't have to be able to emulate native remote instructions that might be accessible via specific REST interfaces of the TV.

Current deployed HbbTV applications mostly run on an unsecured HTTP connection on port 80. Even though it should be possible to also support apps on different ports with secured SSL connections it is not considered supporting that out of the box. However since this component allows to inject a script it should be possible to use this driver to run tests on pages that are manually instrumented via the launcher script.

3.2 Nonfunctional Requirements

In the end both components should allow to run a cloud of instrumented Smart TV devices within the same network. It should not require any human interaction with the TV to run automated tests on them or debug the HbbTV app. If the DevTools Backend is used as a proxy on a peripheral device like a Raspberry Pi it should self heal and bring itself back up in case of a severe error. Using a preconfigured Pi it should be a simple plug & play configuration to set up a new TV. That allows to simply scale up the number of connected Smart TVs to the Selenium Grid.

The Chrome DevTools is an already known tool already used by many developers. By adapting the Remote Debugging Protocol and providing the exact same user interface to debug and inspect HbbTV applications makes this solution really easy to work with. Moreover it allows to also integrate with other tools from the web ecosystem.

At the end both components need to be delivered with proper documentation of its functionalities as well as decent test coverage. Developers at the Fraunhofer Fokus should be able to install, build and develop the components based on documentation within the repositories. Step by step instructions shall allow everyone to set up an additional TV to the existing grid or setup a complete new grid in a TV lab.

3.3 Technical Requirements

To run a grid with instrumented TV devices it is required that these devices are HbbTV compatible. The device itself should not be turned off at any time as it is not possible to

[5] `https://github.com/appium/appium-xcuitest-driver`
[6] `https://github.com/appium/appium-android-driver`

turn them on again. The DevTools Backend is supposed to run as proxy on a Raspberry Pi 3 model B that runs the latest Raspbian[7] operating system. The Chrome DevTools is due its nature only supported on Chrome browser. Even though you can instrument any web page running in any kind of web environment the DevTools application will be only accessible with Google Chrome[8]. The Raspberry Pi requires an ethernet to USB adapter in order to manage two connections. One from the TV to the Raspberry Pi itself and the other from the Raspberry Pi to the network. If not available it is also possible to use the Wifi module to connect.

To simplify the developing process of both modules it should be straight forward to build and run the project. All steps have to be documented and simplified in a way that it can be run on any desktop OS.

3.4 Social Requirements

As mentioned in section 2.2 e2e testing is not easy. Due to its nature it is not the most popular form of software testing. Therefor both components should ensure that the testing experience is less brittle and flaky. The HbbTV driver should consider the low CPU power of a standard TV and should behave responsive when an HbbTV app gets loaded. Tests have to run stable and don't result in false positives. Since there are already other companies with a similar approach of HbbTV testing this tool should demonstrate its advantages easily with its integration to the Chrome Remote Debugging protocol. However this is not much worth if the driver responds inconsistent or is difficult to setup. At the end the developer should feel some kind necessity to use the tool as it makes him more productive. Within a team and project it should clearly improve the release cycles and the confidence that the produced code has no bugs and unexpected regressions.

[7] `https://www.raspberrypi.org/downloads/raspbian/`
[8] `https://www.google.com/chrome/index.html`

4 Concept

This section will explain in more detail the concepts of each individual component and how they will play together. They individually contribute to the overall goal of implementing a development and test automation platform to build HbbTV applications. The DevTools Backend service will be the base component and will help us to achieve the first part of building a state of the art web-authoring experience. It enables new possibilities for developers to inspect an HbbTV app and understand and debug JavaScript problems within the code. It helps to instrument the application and also to run automated WebDriver tests with the Appium HbbTV Driver. This component acts as translator between the WebDriver protocol and the Remote Debugging Protocol. To do that on a bigger scale we need the Raspberry Pi to deploy that driver to any TV without any manual steps. To manage all these drivers we then use a Selenium Grid and with a proper CI/CD server we can leverage that setup to test and release our HbbTV apps faster and with more confidence.

4.1 Components

4.1.1 DevTools Backend

The DevTools Backend is the main component of the overall design. It has to do most of the work and is the only component that directly interacts with the targeted environment: the HbbTV application. Debugging an HbbTV application these days is almost impossible. Even though some TV manufactures provide some interfaces and APIs to connect to the TV they are barely documented and almost different for each TV model. To provide a tool that covers all TVs of all manufactures, it requires a different approach than hooking into a native interface. Until all manufactures recognize the demand for developers to get a better development support this won't change. We already had a similar situation a couple of years ago when the smartphone market started to explode and a lot of people started writing mobile or hybrid apps for Android and iOS. At that time both vendors had almost no support for any debugging tools that would help developers to inspect the page. A tool called *Weinre*[1] was developed and found a lot of popularity within the developer community. It enabled to debug arbitrary web pages remotely for the first time, especially on a mobile device such as a phone. It used a unique approach that allowed to do that for all mobile environments (Android, iOS and Hybrid web applications run by PhoneGap/Cordova). After vendors started to support

[1] WEb INspector REmote - `http://people.apache.org/~pmuellr/weinre/docs/latest/Home.html`

remote debugging natively due to the success of this project it became obsolete and the inventor stopped maintaining it.

This project is the role model for the DevTools Backend component. Similar how it allows to remote debug applications for smartphones, the DevTools Backend will allow it for HbbTV based apps on Smart TVs. The idea is simple. A script that gets executed within a targeted environment connects to a server to exchange information and commands as utility for a 3rd party authoring tool. The concept works independently from the environment and device. As long as the target supports basic web technology like HTML and JavaScript it will run everywhere. This component can not only be used to debug HbbTV apps on Smart TVs but also to inspect any other IoT device like a fridge or a coffee machine as long as their interfaces are based on web technologies[2] [3].

The component itself consists of two subcomponents. Similar to *Weinre* it has a frontend part that takes care on instrumenting the target environment and a backend part which is a server that initializes and manages the data traffic between target environment and authoring tool. Both subcomponents have logic to handle methods or trigger events according to the Remote Debugging Protocol. As stated in section 2.3 the Remote Debugging Protocol is supported by all WebKit browser and has first class support for one of the most used web authoring tools these days: the Chrome DevTools. In addition to that it is actively maintained by a dedicated team at Google and is well documented[4]. In order to provide as much integration without any additional effort it only makes sense to rely on a well maintained protocol like this.

When it comes to debugging an HbbTV application there will be always three parties involved. The instrumentation logic, the backend and the authoring tool. The instrumentation logic is what actually interacts with the target environment. It executes commands that it receives from the backend and returns desired information. Since the instrumentation script is not much more than any other script on the page it can't support all methods that are defined within the Remote Debugging Protocol. However the JavaScript API is pretty comprehensive and covers the essential requirements of that protocol like returning and modifying the state of certain properties like e.g. DOM nodes. Many things can be emulated which is sufficient for this use case. It is important to ensure that the instrumentation script doesn't affect in any way other scripts or the web application in general from behaving differently. It has to keep its environment as pristine as possible. As a debugging tool it should help developers to find bugs and not create them. Once an instrumentation script was injected into the environment it has to register itself to the backend. Script and backend establish their connection via WebSockets. Since events and methods will be exchanged randomly from both sides it is important to have a full duplex connection type so both ends can communicate with each other in an asynchronous fashion. To identify the target, the instrumentation script has to identify itself with a unique id that can be either the host of the HbbTV app or a randomly chosen

[2] In fact the screen in the Tesla Model S is build on top of a proprietary web browser and therefor all Tesla apps are build with web technology. Theoretically the DevTools Backend can be used to debug these apps with modern web authoring tools already today.

[3] An example application can be found here: `http://dash.time4tesla.com/`

[4] `https://chromedevtools.github.io/devtools-protocol/`

string. The backend will register the page based on that id and some other metadata and will manage any connections to it so that arbitrary clients can connect to the page with the backend as proxy. Due to some house keeping work the backend will never directly connect a client to the instrumented page. In fact it will manage the communication with two different WebSocket channels. Not all method requests from clients are directed to the page itself. For example all methods of the Remote Debugging Protocol that relate to network events or page lifecycles are handled by the backend. Figure 4.1 shows the activity between all parties in more detail.

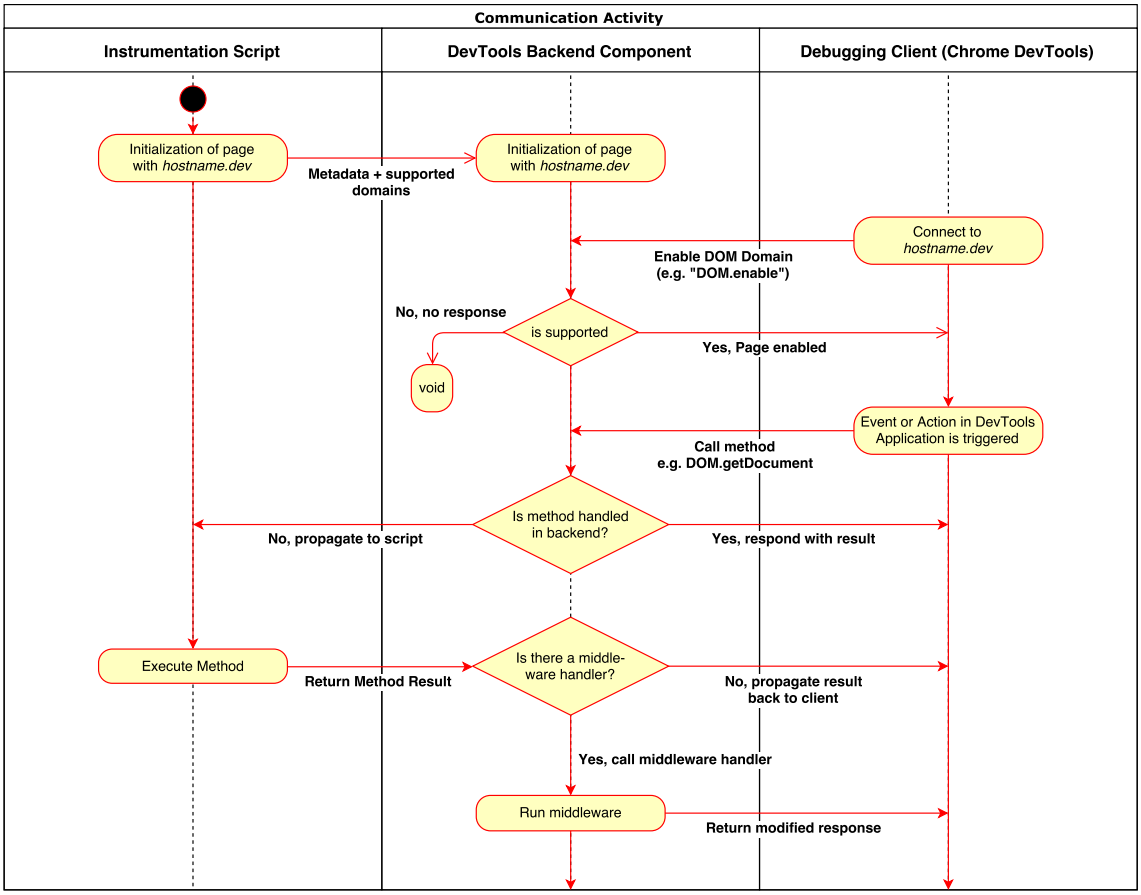

Fig. 4.1: Communication Activity between Instrumentation Script, Backend and Debugging Client

In scenarios where a full page load happened the instrumentation script looses the connection with the backend. It manages to reconnect to the target, given that the page unique id is still the same, and makes sure it propagates the right lifecycle events to the connected clients. They won't recognize any connection abruption and instead properly handle the page load on their side. In some cases the response of the instrumentation script needs to be enhanced with data from the backend. This happens when network

data is involved which is only stored on the backend side. A middleware handles this situation by enhancing the result object before returning it back to the client.

There are two ways to inject the instrumentation script into the target environment. It can be either placed manually into the page by referencing the script via script tag or the DevTools Backend can be used as a proxy. In this case it captures all http packages on a certain port (usually port 80) and injects the script inline into the page automatically. The proxy registers the page to the backend based on the data it gets from the request. This has the advantage that the application doesn't have to get modified to instrument it. However running the target device (e.g. the Smart TV) through a proxy is not always possible. Therefor the component provides a launcher script that can connect to the backend to register the page before the instrumentation script gets initiated.

Once this happened and a remote debugging client connects to the backend they both exchange JSON payload over the socket channels. The client usually listens to certain domain events (depending on the use case) and fires methods with an id, a method name containing the domain and the request (e.g. `CSS.getMatchedStylesForNode`) as well as some parameters that get propagated to the method (e.g. a node id). Each domain has to get enabled by the debugging client so that the communication can be reduced to the minimum. If the client would enable all domains it would overflow the socket channels. This keeps the data amount manageable. The backend only propagates messages to the instrumentation script if the containing method was enabled by the client beforehand. Even though all methods and events of the protocol are documented in a comprehensive way[5] the order of events and methods that have to get triggered on both sides is unknown. The only way to find out when certain events has to get thrown is by debugging the debugging tool. Since the Chrome DevTools is just a normal web-application it is possible to debug it using the Chrome DevTools itself. Within the *Timeline* tab it is possible to inspect the WebSocket connection to the browser in real time. This allows to reverse engineer the protocol and emulate a similar behavior like in the browser.

4.1.2 Appium HbbTV Driver

Based on the DevTools Backend component the Appium HbbTV Driver is now able to run its automated tests without having to take care about how to automate WebDriver commands on the TV. Since the DevTools Backend is applicable to all HbbTV supported devices the driver can therefor be utilized on arbitrary TVs or SetUp boxes supporting that format. It uses the already existing framework utilities provided by Appium to support all common functionalities that an automation driver requires. Appium already provides a variety of drivers that are mainly focused on mobile automation. However its vision is to create more drivers that support new devices and platforms. HbbTV is the perfect fit for that. With Appiums Basedriver module[6] it provides not only a template to build the driver but also comes with all required features to run it on bigger scale e.g.

[5] see `https://chromedevtools.github.io/devtools-protocol/`
[6] `https://github.com/appium/appium-base-driver`

within a Selenium Grid out of the box. An overview about all involved components is shown in figure 4.2.

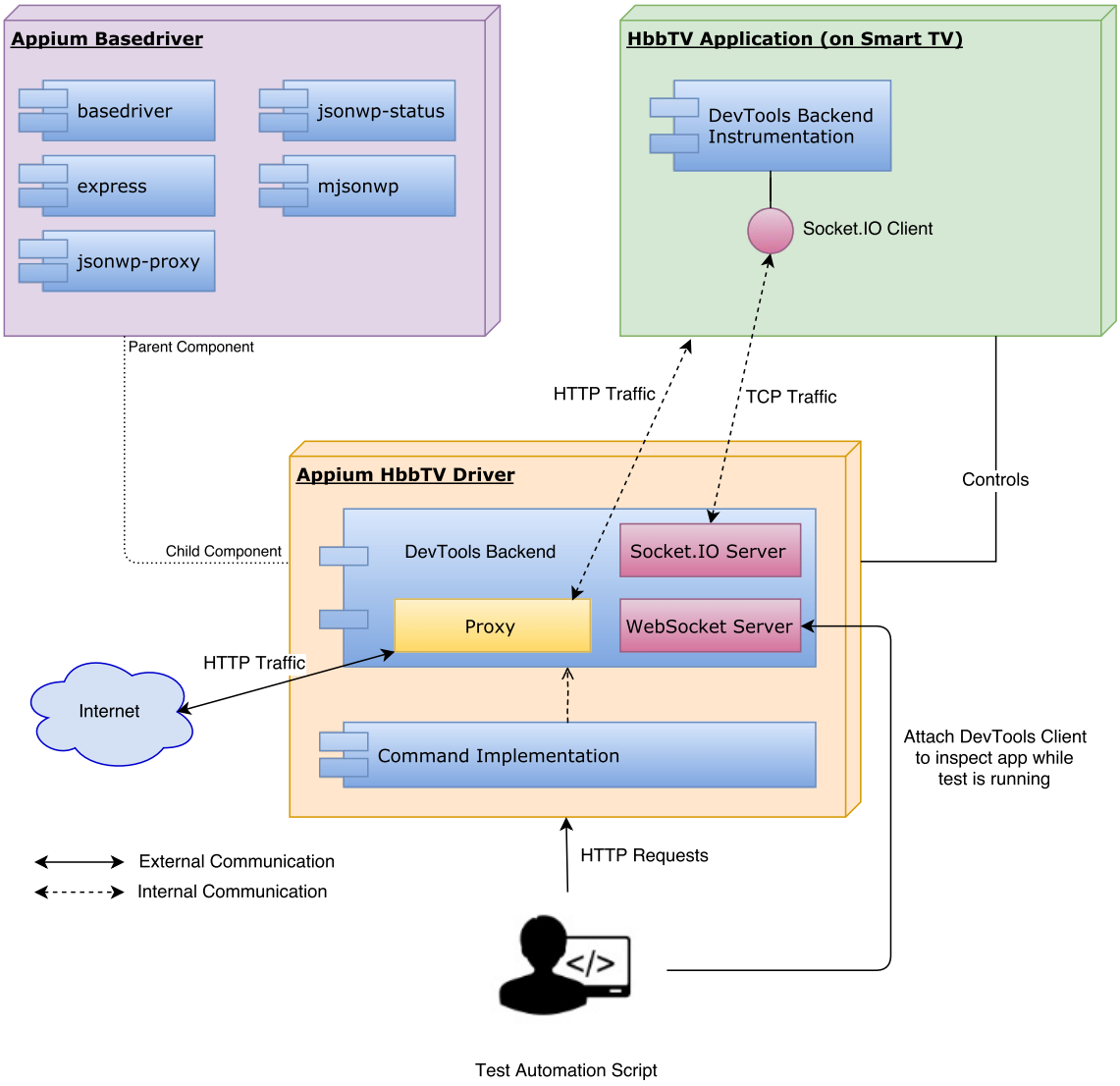

Fig. 4.2: Appium HbbTV Driver components

Thanks to Appiums Basedriver a lot of important functionality can be inherited. It builds the foundation of the actual driver. It takes care of session management and capability definition/validation (*"basedriver"*), provides predefined WebDriver routings (*"mjsonwp"*) which are served by a preconfigured HTTP server component (*"express"*).

With that it manages incoming HTTP requests and handles their responses by applying the correct response codes (*"jsonwp-status"*). That also allows additional features like proxying these requests (*"jsonwp-proxy"*). Since it is using the Appium framework on one side and the DevTools Backend on the other the only task of this driver is to translate the WebDriver commands into the Remote Debugging Protocol. WebDriver commands are send as normal HTTP requests. The driver acts as a restful HTTP server and can receive these request. Depending on the endpoint it triggers a different action on the target device. Every automated test starts by initating a WebDriver session. Figure 4.2 demonstrates this process in detail.

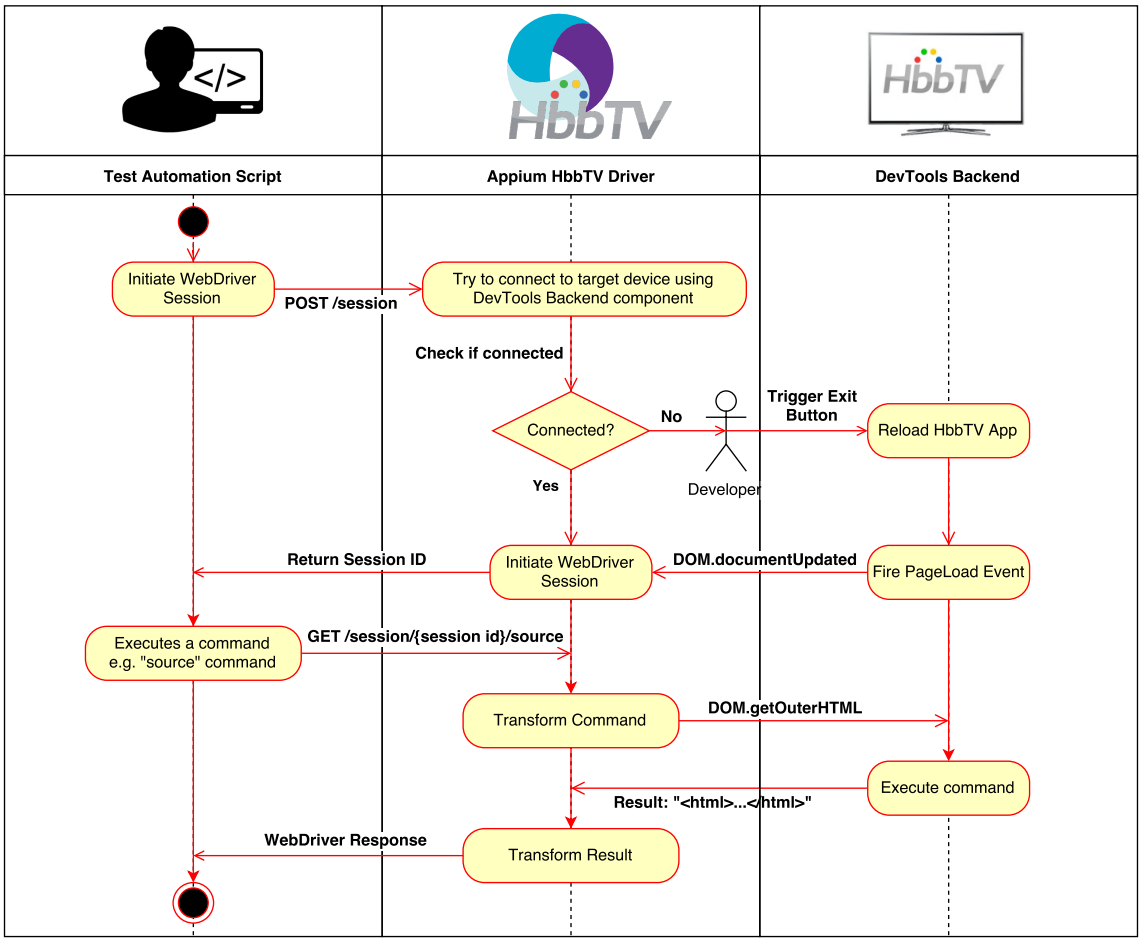

Fig. 4.3: Creating A WebDriver Session

A session can only be successful initiated if the DevTools Backend component is able to successfully connect to the target device. In case the TV is turned off or never got

instrumented by the DevTools Backend it requires one manual step to finish the initiation process. By pressing the Exit button on the remote the HbbTV app gets reloaded and the DevTools Backend can inject the script if used as a proxy. If the instrumentation script is injected manually the test script can connect to it by providing the page ID within its capabilities. Once a connection to the target device was established the initial session request can be resolved. The response includes a session ID next to other information of the assigned device. The session ID is important as it is part of almost all WebDriver endpoints to identify the user and execute all commands on the right device.

Depending on the test framework of the developer the session ID gets automatically stored within the test context. The developer can now run any WebDriver commands. After the driver has received a new command it transforms it into a Chrome DevTools Protocol method and forwards it to the DevTools Backend component. Since the DevTools Backend is using an asynchronous socket connection the driver has to wait until either the result of the method was returned or a certain event was triggered in order to resolve the WebDriver HTTP request. Certain commands can trigger a page load in the target environment that will break the connection between the DevTools Backend component and the instrumentation script. In these cases the driver has to wait until a reconnection happened in order to resolve the request, otherwise the test script would not be able to send the next command. If the connection can't be re-establised a timeout will be hit and causes to return with an error.

4.1.3 Raspberry Pi

To ensure that such connection loss never happens a man in the middle component between network and Smart TV can be used. A peripheral like a Raspberry Pi[7] is an ideal utility to manage a TVs network traffic and instrument any HbbTV app on the fly. A Raspberry Pi is a *"fully customizable and programmable small computer board"*[Maksimović et al., 2014] with a size of a credit card. It has a 1.2GHz 64-bit quad-core ARMv8 processor with 1 GB RAM, 4 USB and 1 Ethernet port. It is known as the mainstay in the world of makers and electronics but also as an educational device that brings people the joy of electronics and computer programming. With 38€ it is super cheap and yet powerful. People who are familiar with Linux have no problems to start building tools and applications with it. With a prebuild Raspbian operating system the provisioning of a TV to test HbbTV apps can be reduced to a simple plug&play step.

The idea is that all network traffic of the Smart TV runs through the Raspberry Pi. Running the DevTools Backend component it acts as a Proxy between TV and network. This has the advantage that every HbbTV app will be instrumented on the fly. There is no need to manually inject an instrumentation script. The DevTools Backend proxy modifies every HTTP package that has an HbbTV mime type before it sends it back to the TV. In addition to that it tracks all application assets so that the developer can inspect all files that have been downloaded by the HbbTV app including information like request and response headers. This setup allows to not only do that with own applications

[7] https://www.raspberrypi.org/

but also with any HbbTV app of any broadcaster that can be received by the Smart TV. Having said that, it allows to monitor all communications between HbbTV app and server and helps to understand how certain applications track your behavior while using the app.

In order to put this together, it requires an additional ethernet cable as well as an USB to Ethernet adapter. The Raspberry Pi has to maintain two network interfaces. One is for the data transfer between TV and Pi and the other for the one between Pi and network. Therefor you could theoretically also use the Wifi module to connect to your network instead of using the USB adapter. With the correct network setup it should connect automatically. Once it is connected to the power supply the operating system boots up and starts the Appium HbbTV Driver. To identify the Raspberry Pi we change the hostname settings to be the model name of the TV it is connected to. After the Pi has established a connection between TV and our network we can immediately address this host as our automation server and run tests on it.

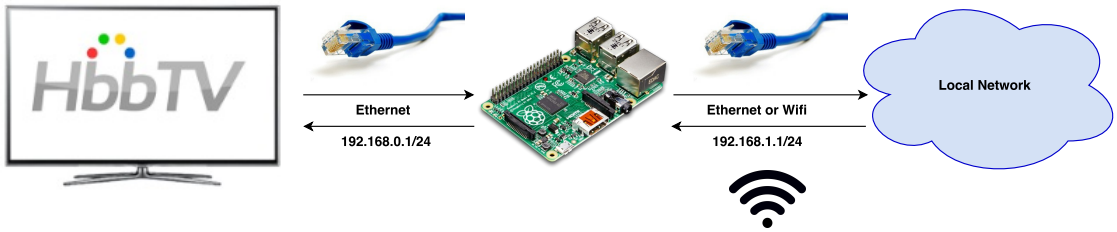

Fig. 4.4: Setup of SmartTV with Raspberry Pi

To seperate the network traffic we have to run both interfaces on different subnets. This can be customized depending on the given network. Most local networks will have an IP range of `192.168.1.1/24` which is why this setup would make the most sense as default. However once a proper network is found it is possible to copy the images to other Raspberry Pis. This will allow us to provision an already existing TV lab in a simple way and deploy our components to run automated WebDriver tests using the Appium HbbTV Driver and DevTools Backend.

4.2 Selenium Grid

To run our automated tests on multiple TVs with different HbbTV support at the same time we need a Selenium Grid to orchastrate the WebDriver requests. The Appium HbbTV Driver component allows us to register to such a grid using an internal settings page that the user can open in the browser. The only information that is required is the IP address of the Selenium Grid server. That server has to be accessible in the network where the TV and the Raspberry Pi are located in. The driver will send a request to

the Selenium Grid server containing the capabilities of the TV. In order to receive those from the device it is important that the DevTools Backend is able to connect at least once with an HbbTV app to identify the TV by looking on its user agent. It always constains the supported HbbTV version as well as the name of the manufacturer and device model. If the registration to the grid server was successful it will automatically ping the Raspberry Pi every once in a while to check its connectivity and status.

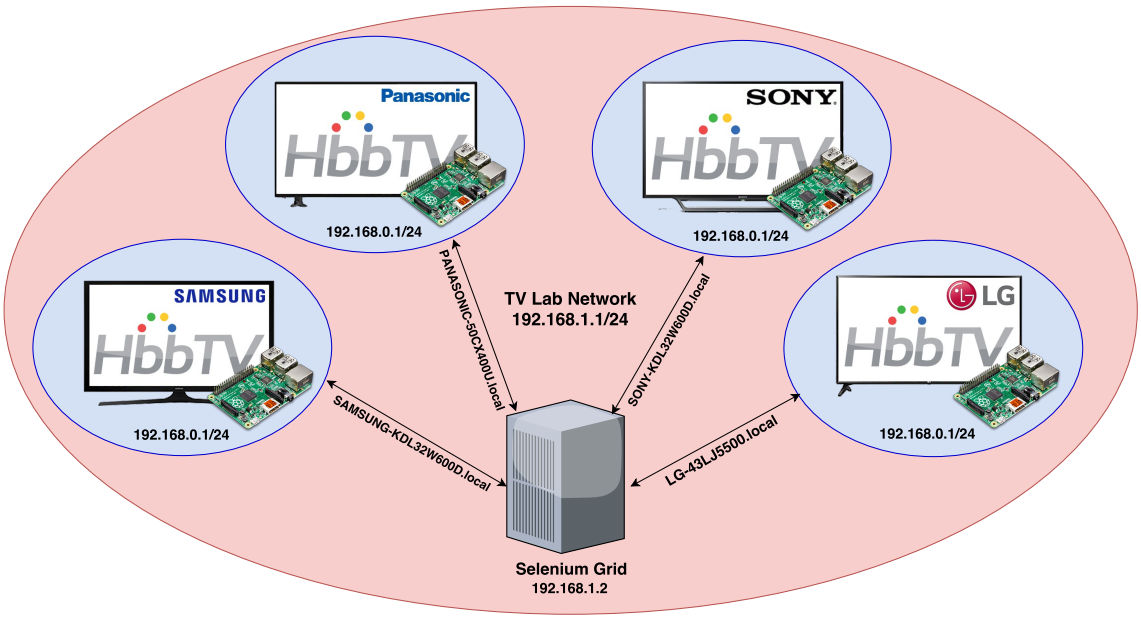

Fig. 4.5: Network Setup of TV Lab with Raspberry Pi

The Selenium Grid will be the access point for all developers to run their automated tests on. It manages its capabilities and only creates a session if the desired device is available and not used by someone else. If a device is blocked due to another test it will queue up the request and waits until it becomes available again. Each Smart TV is registered with the supported HbbTV version, the name of the manufacture and its model name. Each test can therefor specify on what kind of device it should be running. The Selenium Grid tries to find the best suiteable device if available. If for example the developer wants to run a test on a Smart TV with supported HbbTV 1.5 the grid server randomly selects a Smart TV that runs that HbbTV version. The same goes for other capabilities. All data about registered Smart TVs and their availability can also be consumed as JSON blob using the Rest API of the grid server. This allows creating custom dashboards that are more aligned to the TV theme. Instead of just model names it could display the TV model as image with the manufacture logo.

4.3 Continuos Integration and Delivery

The last step to have a fully automated continuous integration and delivery pipeline is to setup an automation server that runs our tests as soon as any code changes were introduced to the project. There are multiple ways to setup such pipeline. CI/CD became a major part in the software delivery process over the last years so that many tools have been developed since then. One of the most popular open source solutions is Jenkins[8]. It is *"deployed at large scale in many organizations"*[Berg, 2012] and *"has no licensing costs"*[Berg, 2012]. In combination with a source control tool like Git[9] we can trigger our automated tests regularly so that defects are detected at early stage before it is deployed to production. Depending on the project and its requirements you can deploy different strategies on how code gets tested and pushed to a production environment. This can be fully automated so that no human interaction is necessary anymore.

[8] `https://jenkins.io/`
[9] `https://git-scm.com/`

5 Implementation

Since we are working in a web environment it only made sense to build all software components with a popular web technology too: JavaScript. Both the DevTools Backend and Appium HbbTV Driver components are written in NodeJS using a Babel[1] compiler to use the latest EcmaScript[2] features today. They both use Eslint[3] for static code analysis and have a Dockerfile so they can get seamlessly deployed on any system that has Docker[4] installed.

5.1 DevTools Backend

The DevTools Backend has a simple NodeJS project structure with a *package.json* that contains a handful of CLI commands to operate and build the project. It is stored on the Fraunhofer GitLab server and can be cloned with git if permissions are granted. Once it was cloned you can simply install the dependencies and build it using the NPM commands. The project requires to run a NodeJS version of v7.4.0 as defined in *.nvmrc*.

```
# clone project
$ git clone git@gitlab.fokus.fraunhofer.de:christian.bromann/devtools-backend
# install dependencies
$ cd devtools-backend
$ npm install
# build project
$ npm run build
# run server
$ npm run start
```

Listing 4: Setup DevTools Backend component locally

With Docker this procedure is simplified down to these commands:

[1] `https://babeljs.io/`
[2] Ecma is a standards organization that defines the ECMAScript language which was created to standardize JavaScript
[3] `http://eslint.org/`
[4] `https://www.docker.com/`

```
# clone project
$ git clone git@gitlab.fokus.fraunhofer.de:christian.bromann/devtools-backend
# install dependencies
$ cd devtools-backend
$ npm run docker:build
# run container
$ docker run -p 9222:9222 devtools-backend
```

Listing 5: Setup DevTools Backend component with Docker

The build process not only compiles all server side files down to ES5 using Babel, it also compiles all frontend related files (the instrumentation code) to one JavaScript file which gets injected into the HbbTV app. This has the advantage that the structure and the way these files are written don't differ from each other. Thanks to WebPack[5] all frontend files are bundled together so that they can get executed in the browser. This makes it really easy to structure and share code on server and frontend side. The main server is build on top of Express[6], a minimal and flexible web framework to build web applications in NodeJS. It allows to define API endpoints and provides useful utility features to work with Cookies and caching. All application code can be found within the *lib* directory. Next to the main server and proxy logic most of the code is split up in backend and frontend. The backend part is responsible to route all WebSocket communication between the remote debugging client (e.g. the DevTools application) and the frontend automation code that gets executed on the Smart TV. It also tracks all network traffic that comes through the proxy and returns the data to the client if desired. All backend related information like registered pages, resource content or metadata are being held there to save the request to the frontend. For simplicity reasons this data is held in memory.

As *"page"* we define a debugging target that can be either an HbbTV app on a SmartTV or any other web page where the instrumentation code is injected and connected to the backend. The server can be used to debug multiple pages at the same time. Each page has a unique identifier that allows to connect with a client to the desired page. The communication between the debugging target and the server gets initiated via Socket.io[7]. Socket.io is a sophisticated NodeJS module that enables real-time, bidirectional and event-based communication between server and client. It is not only really fast and efficient, it also works reliably on many web platforms. Especially on Smart TVs where not all devices have support for WebSockets, Socket.io ensures that a connection can be established by using fallback strategies like XHR polling. From the server to the debugging client a standard WebSocket server architecture was chosen since there are no such fallback requirements to be fulfilled. Every time the proxy injects an instrumenta-

[5] https://webpack.js.org/
[6] https://expressjs.com/
[7] https://socket.io/

tion script or the launcher registers itself to the backend a new page instance is created and two new Socket channels are opened.

```
export default class Page extends EventEmitter {
    constructor (io, uuid, hostname, url, title, description, metadata) {
        super()
        // ...

        this.io = io.of(`/page/${this.uuid}`)
        this.io.on('connection', ::this.connect)

        this.wss = new WebSocket.Server({
            perMessageDeflate: false,
            noServer: true
        })
    }
    // ...
}
```

Listing 6: Socket Channels initiated in Page class

It is worth mentioning that the backend doesn't create a WebSocket server every time a page instance gets initiated. It instead attaches itself to the Express server and creates a fake server instance (see *"noServer: true"* in listing 6). When a client asks for a WebSocket connection an *"upgrade"* event is emitted when the opening handshake takes place. It *"upgrades the connection from HTTP to the WebSocket protocol"*[Fain et al., 2014].

```
// lib/index.js
export default class DevtoolsBackend {
    constructor (host = DEFAULT_HOST, port = DEFAULT_PORT) {
        this.app = express()
        this.server = this.app.listen(port)
        this.server.on('upgrade', ::this.backend.upgradeWssSocket)
        // ...
    }
    //...
}
```

Listing 7: Server Initiation with ExpressJS

The backend module keeps a list of all registered pages and can upgrade the connection properly if the event was fired by the server. Depending on the page id, which is an

uuid, the backend connects the debugging client to the desired page and enables the communication between client and instrumentation code.

```
// lib/backend/index.js
upgradeWssSocket (req, socket, head) {
    const pathname = url.parse(req.url).pathname
    for (const page of this.pages) {
        if (pathname === `/devtools/page/${page.uuid}`) {
            return page.wss.handleUpgrade(
              req, socket, head, ::page.connectWebSocket
            )
        }
    }
    socket.destroy()
}
```

Listing 8: Multiple Socket Channels Registered on one Server

Once the connection was established the debugging client can send commands and receive events from the instrumentation code. In most cases the backend only propagates these messages from Socket.io to the WebSocket connection and vice versa.

5.1.1 Instrumentation Script

The frontend part of the DevTools Backend component is the instrumentation script. It gets executed in the target environment, e.g. the HbbTV browser on the Smart TV. It connects to the backend via Socket.io and is used to propagate page events to the debugging client as well as execute commands to transform elements or get information about the page state. The instrumentation script should be placed at the top of the `<head>` block to ensure that it gets executed before everything else. This is important because it has to capture as many events as possible. In case the HbbTV app fails to load due to a JavaScript error it only can propagate that error to the debugging client if it has registered an error handler before the HbbTV code was executed. It also overwrites certain APIs like the console object to intercept messages and events. All methods for a certain Chrome DevTools Protocol domain are stored in one file. Listing 9 shows an example of an implemented method[8] for the page domain. As mentioned above it is written using the latest EcmaScript language features like the destructuring assignment syntax[9] or the export statement[10]. Even though these features aren't supported in current browsers (and especially not in proprietary HbbTV browsers of old TV models) they can still be used since the code gets compiled down to a legacy JavaScript version.

[8] https://chromedevtools.github.io/devtools-protocol/tot/Page/#method-navigate
[9] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
[10] https://developer.mozilla.org/en/docs/web/javascript/reference/statements/export

This ensures that the code is fully compatible on the target device. However this doesn't replace missing JavaScript APIs. Therefor any APIs that have not been defined within the EcmaScript 3 specification, which is the supported version for HbbTV 1.0, are either avoided or used in a way that any instructions would fail silently.

```
/**
 * Navigates current page to the given URL.
 *
 * @param  {String} url  URL to navigate the page to.
 */
export function navigate ({ url }) {
    if (typeof url !== 'string') {
        return
    }
    window.location.assign(url)
    return {}
}
```

Listing 9: "navigate"Method of Page Domain

All domain logic is then imported into a single module to export it as central API. This pattern is called *"Barrel"* and has its origin from the TypeScript world. *"A barrel is a way to rollup exports from several modules into a single convenient module. The barrel itself is a module file that re-exports selected exports of other modules"*[Syed, 2017].

```
import * as CSS from './css'
import * as DOM from './dom'
import * as Debugger from './debugger'
import * as Input from './input'
import * as Network from './network'
import * as Overlay from './overlay'
import * as Page from './page'
import * as Runtime from './runtime'
import * as Target from './target'
import * as Webdriver from './webdriver'
export default {
    CSS, DOM, Debugger, Input, Network, Overlay,
    Page, Runtime, Target, Webdriver
}
```

Listing 10: Barrel Module Containing All Domain Logic

This allows us to import one file containing an interface to access the whole domain

logic. With only a couple of lines we can then register a listener to the Socket.io channel for each domain and execute a method based on the message parameters.

```
import domains from './domains'
for (let [name, domain] of Object.entries(domains)) {
    this.domains[name] = domain
    this.socket.on(name, (args) => this.dispatchEvent(domain, args))
}
```

Listing 11: Register Listeners to Socket Connection

The *dispatchEvent* method checks if the action that is defined within the arguments is implemented and executes it if so. It then either returns the result of the function call back to the backend and debugging client or does nothing if no result was returned. This approach allows us to enhance the domain logic without having to register new listeners to the socket connection.

To reference DOM nodes back and forth between instrumentation script and debugging client every node on the page is getting referenced with a node id. As soon as the instrumentation script gets initiated and the DOM tree is rendered it runs through the whole tree and sets an internal flag (`_nodeId`) on every node. In addition to that it saves a reference to that node internally so it can find it based on the node id at all times. If a client fetches the document using `DOM.getDocument`[11] the instrumentation script returns instead of the actual node object a representation of it based on the Node[12] class. This representation contains the assigned node id which allows the client to interact (e.g. change attributes or the content) with that node. In addition to that every instance registers a mutation observer[13] on itself. This allows to listen to DOM and attribute changes on DOM nodes and helps debugging clients to update DOM tree representations in their applications (e.g. Chrome DevTools Elements panel). Other composite objects like promises, generators or maps are handled similarly. With the PropertyObject[14] class the instrumentation script converts each object into a different representation of itself to allow data exchange over the HTTP protocol. Instead of the actual object it returns an object id that is stored in an ObjectStore[15] on the page. Since these objects can't be serialized they remain there and can be referenced using the object id. Similar to how DOM nodes are stored every object in the object store is held in memory. Since every object is passed by reference this won't have any affect on the performance of the page.

[11] `https://chromedevtools.github.io/devtools-protocol/tot/DOM/#method-getDocument`
[12] see lib/frontend/models/Node.js
[13] `https://dom.spec.whatwg.org/#mutation-observers`
[14] see lib/frontend/models/PropertyObject.js
[15] see lib/frontend/models/ObjectStore.js

5.1.2 Launcher

The Launcher script is what's used to manually inject the instrumentation logic on arbitrary platforms without setting up a proxy. It contains next to the Socket.io client library an initialization part and the actual instrumentation script. The initialization part is important as the script needs to register itself as a page to the DevTools Backend component. If a proxy is used this usually happens on the server side. To inject the launcher a script tag has to be placed at the top of the `<head>` block so that it gets executed first. The actual code can be received from the DevTools Backend server.

```
<html>
<head>
    <title>My Target Page</title>
    <script src="http://<devtools-backend>:9222/scripts/launcher.js"
            data-origin="debugger">
    </script>
    <!-- ... -->
```

Listing 12: Inject Launcher Script

Once the script is downloaded by the browser it sends an XHR request to the server with some page metadata (e.g. uuid, url, title, user agent, etc) asking to register this page to the backend. After that it initates the instrumentation scripts and starts the communication to the backend via Socket.io. The uuid can be defined manually by adding a `data-uuid` attribute to the script tag. If this attribute can't be found it uses the hostname as unique id. The `data-origin` attribute is used to hide this DOM node from the debugging client. If a client requests DOM nodes from the instrumentation script these nodes will be ignored so that they don't show up in any DOM representation as they actually are not part of the web application.

5.1.3 Proxy

To automate the script injection the DevTools Backend component provides a proxy that modifies the content of certain HTTP packages. It injects the Socket.io client library and the instrumentation script inline so that it can establish a socket connection with the backend even before the document has fully loaded. Since all network traffic will go through the proxy it is able to track all packages that have been requested by the app. With that information we can provide a full network report for each page load including timeline information and data on when the first package was loaded and when loading was finished. Since you can inspect every network package it allows you to see how certain HbbTV applications load data from the backend as well as how it tracks your behavior on the page.

Since we only want to modify packages that contain HTML code and pass on the other packages, e.g. images or scripts, we have to make sure that we filter every request

depending on its type. This can be done in multiple ways. Just by extracting the file ending you can already identify most of the packages by its type. Additionally some requests have an `Accept` header that contains the expected mime type that is about to be returned by the server. However these informations do not fully ensure that the file that is being requested should be modified or not. Given the fact that modifying a wrong file can break the whole app we need to have a better way to detect file types. Therefor the proxy has another middleware registered that makes an `OPTIONS`[16] request to actually decide whether to pass it forward or to load its content in order to inject the instrumentation script. The HTTP options method can be used to not only detect available HTTP methods, it also returns the content type of the requested resource. Since the response doesn't contain any body payload it is really lightweight.

```
class Proxy {
    // ...
    proxyFilter (req, proxyRes, next) {
        req.target = getFullUrl(req)
        request.head(req.target).then((headers) => {
            /**
             * only proxy resource if content type is an HbbTV application
             * Note: to act as normal proxy (for other apps than hbbtv) it
             * should also allow normal html content-type
             */
            const contentType = headers['content-type']
            if (contentType && (contentType.match(/hbbtv/g) ||
                contentType.match(/text\/html;/g))
            ) {
                // pass forward to proxy
                return next()
            }
            // ...
            // if not HbbTV application pipe request directly into response
            const requestCall = request(opts)
            return requestCall.pipe(proxyRes)
        }, (err) => /* ... */)
    }
    // ...
}
```

Listing 13: Filter Proxy Request based on Response Content Type

By calling the `next()` method we pass on the request to the next middleware which actually handles script injection. This happens only if the content type contains HbbTV

[16] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS`

or text/html though. All other requests are getting logged and directly piped to the response. This only works because the request and response objects are read/writeable streams.

The proxy not only ensures that the target page gets instrumented it also allows custom modifications to the browser behavior. It is often the case that you want to demo a certain HbbTV application without having to setup a broadcast stream that sends an AIT package with the correct address to that app. The DevTools Backend component provides a setup page[17] for that scenario that allows to set up an autostart application. When the Smart TV then asks for an HbbTV app it automatically redirects the request to the given URL. For the sake of simplicity the autostart url is stored on the filesystem in a json file and can be access by a helper function.

```
/**
 * check autoload settings
 */
const { autoload, whitelist } = readConfig().data
const cues = typeof whitelist === 'string'
    ? whitelist.split(',').map((f) => f.trim())
    : []

if (autoload && !cues.find(cue => req.get('host').indexOf(cue) > -1)) {
    this.log.info(`Autoload URL found, redirecting to ${autoload}`)
    return proxyRes.redirect(307, autoload)
}
```

Listing 14: Redirect option to demo arbitrary HbbTV applications

As shown in Listing 14 the redirect only takes place when a comma seperated list of keywords is not part of the actual request host. Given that a lot of HbbTV applications contain subpages and sub applications the developer has to define some keywords that have to be whitelisted. Otherwise you will always enter the given autoload page when trying to access a subpage.

5.2 Appium HbbTV Driver

Similar to the DevTools Backend component the Appium HbbTV Driver is a Node.JS project that is similar structured as the DevTools Backend component. The setup as described in Listing 5 is identical. The driver itself doesn't contain a lot of code. Most of the logic is hidden within its dependencies. As described in section 4.1.2 the Appium HbbTV Driver is using the DevTools Backend component to run its automation. It runs on a Raspberry Pi so that it uses the proxy feature of that component to inject the script

[17] Available at `http://<ip-of-raspberrypi>:9222/settings`

on any HbbTV app automatically. In this case it will automatically connect to the page that was modified by the proxy. However you can also run your Selenium tests on any arbitrary web page that was instrumented by the DevTools Backend instrumentation script. By having a `pageId` defined there that matches to the uuid of a registered page in the backend it can establish a connection to that page to send the DevTools protocol commands. In case the Smart TV has no connection to the DevTools Backend the developer has to manually restart the app so the proxy can instrument the HbbTV application. If this doesn't happen the WebDriver session fails.

Once a connection was established the HbbTV driver connects to the WebSocket server of the page (e.g. the target HbbTV application). Similar to the Chrome DevTools application it will receive a lot of app information that are propagated from the DevTools Backend component. With that it is able to send commands to the actual HbbTV page on the Smart TV as well as receive page data. At this point the WebDriver session could be established successfully and the developer can start to run his first automated commands. The command handling itself depends on whether or not the request triggers a page. If it does the driver has to ensure that the DevTools Backend component was able to successfully reconnect to the page. This only happens if we are able to receive certain events from the component.

```
1 commands.setUrl = async function (url) {
2     this.request('Page.navigate', { url }, false)
3     await waitForEvent(this.emitter, 'DOM.documentUpdated', TIMEOUT)
4
5     /**
6      * implicit pause for page to render
7      */
8     await new Promise((resolve) => setTimeout(resolve, 500))
9 }
```

Listing 15: Implementation example of the setUrl WebDriver command

The example in listing 15 shows how a WebDriver command[18] is translated to a Chrome DevTools Protocol command[19] and send to the DevTools Backend component using an internal helper method called "`request`". From there it gets propagated to the actual HbbTV app where the instrumentation script handles the action (e.g. navigate to a page given by the url in the payload). The third parameter of that function indicates how the driver has to behave after the command is triggered. A truthy value would make the driver wait for the exact response. This is used mainly for commands that actually return a value (like a CSS property[20]). In this case however continuing after the command

[18] As defined in the protocol `https://www.w3.org/TR/webdriver/#go`
[19] See `https://chromedevtools.github.io/devtools-protocol/tot/Page/#method-navigate`
[20] when calling the getCssProperty command `https://www.w3.org/TR/webdriver/#get-element-css-value`

would be problematic since the page is about to open a new HbbTV application. By passing in a falsy value it therefor indicates that the command handles the waiting by itself and we expect no response for that command whatsoever. By calling `waitForEvent` in the next line we now wait until a custom event is propagated by the application. In this case we wait for the `DOM.documentUpdated` event which is triggered *"when [the] Document has been totally updated [and] Node ids are no longer valid"*[Google, 2017b]. The command resolves once this event was received. At the end of the command we implicitly wait for a half of a second in order to give the browser time to render the page. Smart TVs tend to be slow in many situations and the HbbTV browser especially is not always quick in rendering the app. This small timeout stabilizes the execution flow and prevents the driver from running into arbitrary race conditions.

5.2.1 Driver Architecture

Thanks to Appiums ecosystem building a driver is fairly simple. The basic architecture is provided by Appium itself. The device automation is the only part that has to be implemented and differs between all drivers. When the driver gets initiated it imports two important modules from the `appium-base-driver` project which is the main dependency of the driver. It requires a `server` module which is an custom Express server that handles all WebDriver requests according to the protocol. This module provides a function that is responsible for registering all WebDriver endpoints as defined in the protocol. The `routeConfiguringFunction` method which is also imported from the `appium-base-driver` then simply maps[21] all endpoints to a certain instance function defined in the Appium HbbTV Driver. With that all requests to e.g. "`/wd/hub/session`" to initialize a WebDriver session are automatically mapped to a function called `createSession` which then initializies the automation as described in the previous section. The base server not only maps the endpoints to certain functions it also checks if the payload is correct and valid according to the protocol.

In addition to the protocol endpoints the Appium HbbTV Driver also registers additional views[22] for the developer to e.g. register the driver to a Selenium Hub via a simple form. As described in section 2.2.1 a Selenium Hub is a central server that collects multiple drivers in order to have a single access point to all automation servers. This allows us to register many Appium HbbTV Driver connected to different Smart TVs to a central point and access all TVs at the same time through it. It simplifies adding more Smart TVs to a sophisticated TV infrastructure. In order to register a driver manually to a remote hub the Appium HbbTV Driver also has a custom view[23] to receive the node config that is required to register to the hub. Based on the capabilities defined in that config and in the payload of the WebDriver request the Hub will select the driver registered with that config.

[21] The mapping is defined in the appium-base-driver project as well: `https://github.com/appium/appium-base-driver/blob/master/lib/mjsonwp/routes.js`

[22] see settings view at `http://<ip-of-raspberrypi>:4723/settings`

[23] Available at `http://<ip-of-raspberrypi>:4723/nodeconfig.json`

5.2.2 Selenium Grid Setup and Scaling

The described implementation allows to seamlessly setup and scale a Selenium Grid with multiple TVs from different manufactures. All you need is a:

- Smart TV connected to an HbbTV Play Out
- Raspberry Pi, including:
  - 4GB SD card provisioned with a predefined image
  - USB to Ethernet adapter
  - Ethernet cable
- Virtual Machine or another Raspberry Pi running a Selenium Hub
- Router that is connected to the internet and all TVs in a single network

The Ethernet port of the Smart TV has to be wired with the USB to Ethernet adapter of the Raspberry Pi that belongs to the TV. The embedded Ethernet port of the Raspberry is then connected to the router. The image on the Pi is preconfigured so it starts all components automatically. It runs a DHCP server that should assign an IP address to the SmartTV after booting[24]. The Pi itself will then get an IP address assigned by the router. The TV should be then connected to the internet via the Raspberry Pi. The last manual step is to open a broadcast stream with an HbbTV signal so that the proxy on the Pi can instrument the page. From that point we can control the HbbTV browser via any WebDriver client. It makes sense to set the hostname of the Pi to the name of the TV model it is assigned to in order to easier access the right device. You can change the hostname by accessing the Pi via SSH and open the internal configuration menu[25]. If you access the DevTools Backend component on the Pi via opening a browser at e.g. `http://ue46f8090sl.local:9222` you should then see the list of inspectable pages which at that point should be the HbbTV app you just opened. Note that the DevTools Backend component is per default registered on port `9222` (standard remote debugging port) and the Appium HbbTV Driver on port `4723` (standard Appium port). Another static server is registered on port 8080 and gives you a list of logs that are captured from both components.

The image for the Raspberry Pi is preconfigured and has all necessary components installed. You can download it under the following Owncloud link: `https://tubcloud.tu-berlin.de/s/DU6R40CoHvHgVk3`. Using tools like Etcher[26] you can flash it on a 4GB SD card. As mentioned above make sure to SSH into the Pi to configure the hostname. The default hostname is `appium-hbbtv-driver` so you can log into the Pi via "`$ ssh pi@appium-hbbtv-driver.local`". The password is the default Raspberry Pi password ("`raspberry`").

[24] If this is not the case the Raspberry Pi has to be rebooted
[25] via "$ sudo raspi-config"
[26] `https://etcher.io/`

Once you have connected the Pi as described to the Smart TV you can start running automated WebDriver tests on them. There are hundreds of WebDriver client libraries publicly available[27]. If you want to run your test on a specific TV you can just point the client to the corresponding Raspberry Pi in the network. Using a popular WebDriver client in JavaScript called WebdriverIO[28] a simple automation script would look like:

```
1  import WebdriverIO from 'webdriverio'
2
3  const browser = WebdriverIO.remote({
4      host: 'ue46f8090sl.local', // or IP of grid server
5      port: 4723, // or 4444 if grid server
6      desiredCapabilities: {}
7  })
8
9  browser
10     .init()
11     .url('http://itv.ard.de/ardstart/index.html')
12     /**
13      * execute JS on the target device returning the user agent
14      */
15     .execute(() => navigator.userAgent)
16     /**
17      * outputs: "HbbTV/1.1.1 (;Samsung;SmartTV2013;T-FXPDEUC-1102.2;;) WebKit"
18      */
19     .then((result) => console.log(result.value))
20     .end()
```

Listing 16: Simple automation script with WebdriverIO to print out the user agent

As shown in the example you don't need to define any capabilities since the automation driver is accessed directly. Based on that you can build more sophisticated test suites for any kind of HbbTV applications. An example of such test suite can be found in the demo directory in the Appium HbbTV Driver repository[29].

[27] A curated list of can be found at `https://github.com/christian-bromann/awesome-selenium#tools`
[28] `http://webdriver.io`
[29] `https://gitlab.fokus.fraunhofer.de/christian.bromann/appium-hbbtv-driver/tree/master/demo`

6 Evaluation

Looking back to chapter 3 describing the requirements of the final product we will evaluate the result and the usability of it in this chapter. The goal was to create a development and testing environment for HbbTV applications that is comparable with the state of the art of modern web development. Building web applications for the big screen turned out to be very cumbersome since there are no tools that help the developer to understand what is going on on the TV. Common workarounds are self build logging overlays on the developed application which might give information about certain variable states but don't disclose the insides of the app at all. The DevTools Backend component is the first tool that allows HbbTV developers to actually inspect a wide variety of aspects of an HbbTV application.

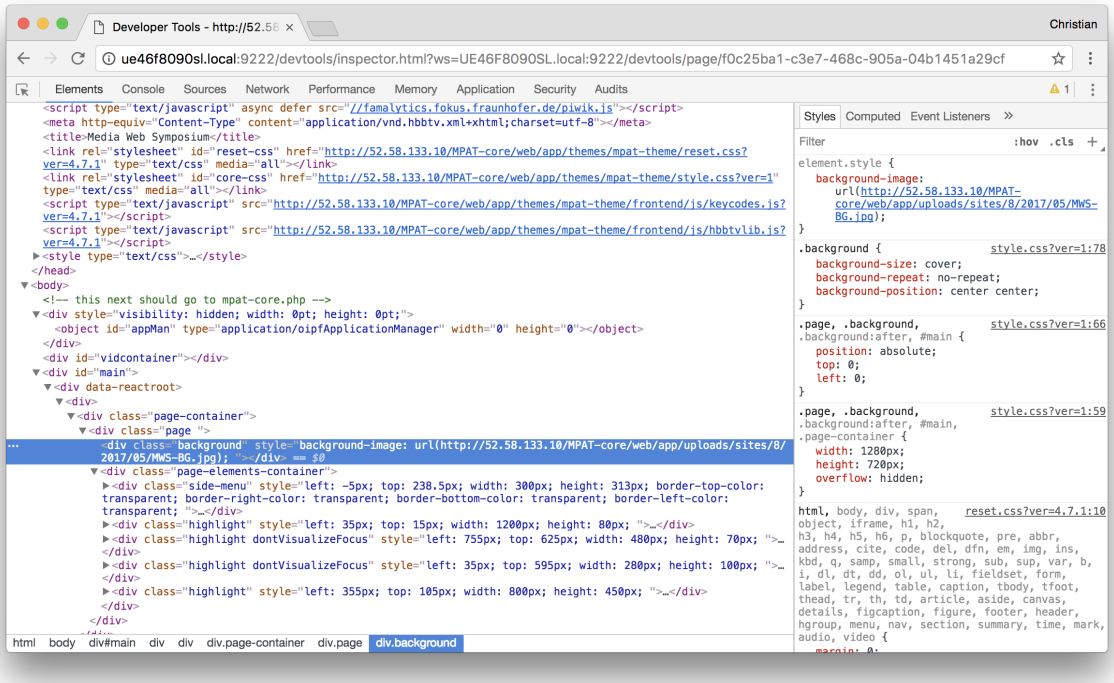

Fig. 6.1: Inspecting the DOM tree of an HbbTV Application using the DevTools application

It not only allows to look into the DOM tree of the app but also to modify elements

and their CSS properties. Developers have now the opportunity to build the app directly on the TV instead of having to implement it on a browser first and then test it on a real Smart TV. Instead of building a custom logging mechanism it automatically captures all logs from the page as well as JavaScript errors that were thrown. In addition to that the *Console* tab of the DevTools application allows to execute any random JavaScript code within the context of the HbbTV page. With that it can inspect variables of your application during runtime. It can be used by the developer to see which JavaScript APIs are available in the browser environment of a specific target device.

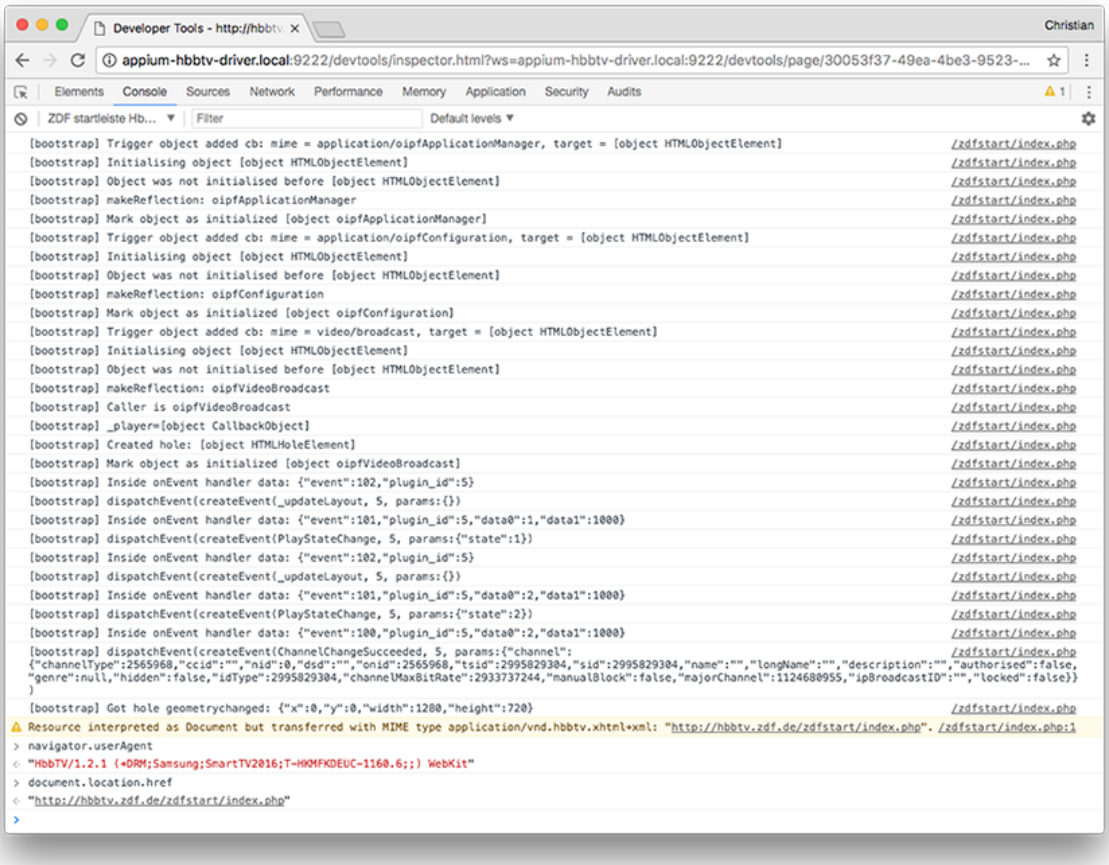

Fig. 6.2: Debugging the ZDF HbbTV application with the Console tab

In addition to that since the TV is running all its network traffic through the proxy on the Raspberry Pi it automatically collects all network data in a way that it can be displayed in the DevTools application as well. It enables developers using this tool to not only see if all network requests have been resolved successfully on their own app but also on any arbitrary HbbTV applications that are published. This gives developers and researchers the chance to look into the loading behavior of an app to reveal information

on when certain data is loaded and if the app is tracking the viewer behavior in any way. Like in the browser the DevTools application as seen in figure 6.3 shows not only a list of all network requests that have been made by the app but also their content.

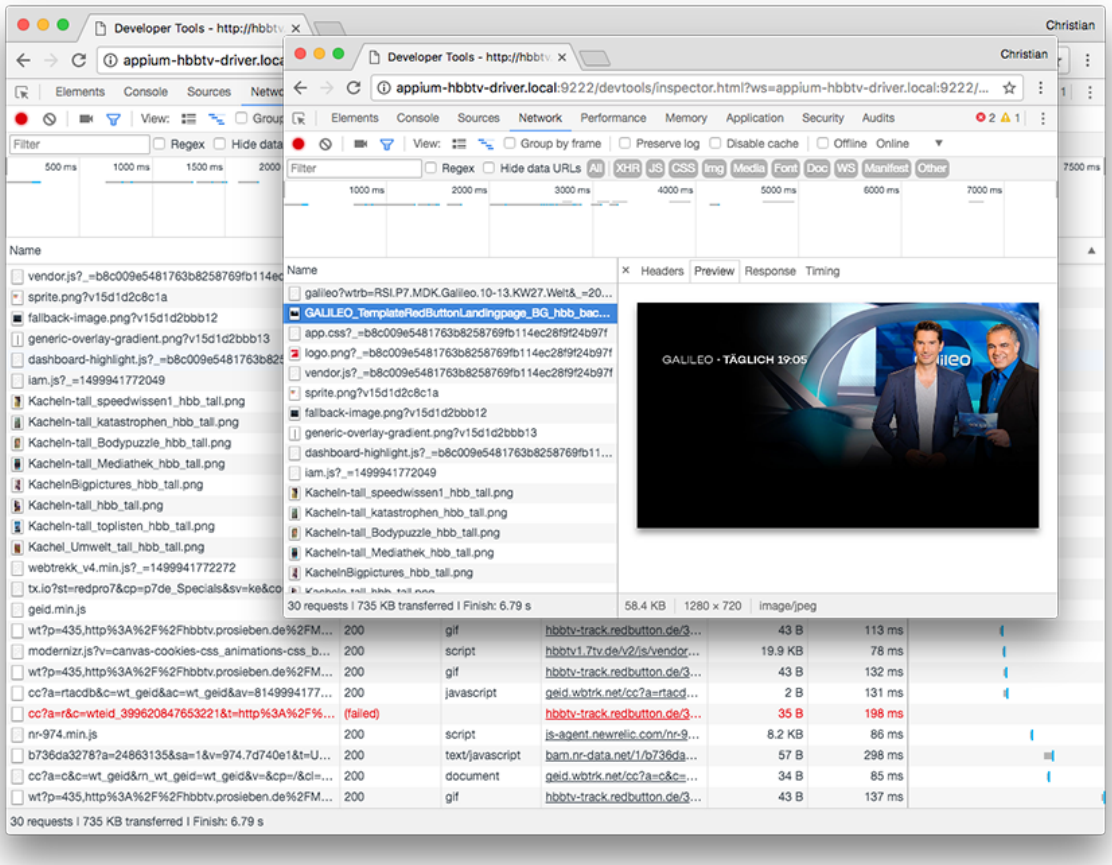

Fig. 6.3: Network requests being made by the Pro7 HbbTV application

Sniffing through the requests of the Pro7 HbbTV application it shows that many small data packages that are covered as 1px gif images are sending user information to services like INFOnline[1] or a New Relic aggregator[2] containing data on the viewer origin, his TV, its model name and other device metrics. Another interesting artifact of data can be observed when switching around pages on the Pro7 HbbTV app. Every time a new page is opened a request is send to `hbbtv-track.redbutton.de` that tracks the movement of the viewer[3]. After the application has fully loaded you can see at the bottom of the page that the main page took about 7 seconds to fully load and it downloaded 735 KB

[1] See request details in Annex section under listing 2
[2] See Annex section under listing 3
[3] Also shown in Annex section listing 4

in 30 requests. This can be used to leverage interesting metrics of the performance of the HbbTV application.

Not only on the debugging side fulfills this tool all requirements that were stated before. Also its testing features raise above everything that has been done in the HbbTV industry before. Until now *"most TV device browsers don't support WebDriver"* [Campbell, 2015a][4] but with the Appium HbbTV Driver developers can for the first time set up a grid of Smart TVs to run WebDriver tests on them. Since it is based on the WebDriver protocol which is the industry standard for automated desktop and mobile testing there are already hundreds of libraries available that can be used to write these automated tests. These libraries are available in all programming languages and flavors and already known by many developers that have experience with automated tests. As shown in listing 16 an automation script can be written within a couple lines of code. These tests are fairly simple to write even for non developers. There are solutions like Cucumber[5] that allow to abstract away the coding part of these tests by writing feature files with acceptance sentences which are translated into actual code. This makes writing e2e tests accessible for everyone.

6.1 Automated HbbTV Test in CI/CD

At the end of the thesis a project was setup that should simulate a continous delivery pipeline for a real HbbTV application. The goal was to create an automated process that tests the app everytime a change was about to get introduced as e.g. a pull request in a project that is managed with a version control system like Git[6]. For this reason a Jenkins server was deployed in the TV lab at the Fraunhofer Fokus. Like in other common pipeline models a job within Jenkins should get triggered once a patch was proposed to the project. Depending on the result of that test it should approve or reject the code change within the version control system. To accelerate the test execution a Selenium Grid server was setup to allow running tests in parallel on multiple TVs. Three different TV devices were equipped with a Raspberry Pi running the DevTools Backend and the Appium HbbTV Driver on them. All drivers got registered to the grid server so that each test is properly directed to the right TV depending on its capabilities. For simplicity reasons the test project[7] only contained the test files and not the app itself so that an already deployed HbbTV[8] application can be used as guinea pig. To show that running functional tests with WebDriver is language agnostic the same test suite was written in 3 different programming languages: Python, Ruby and Node.JS. It contained the following tests:

[4] The tool that is stated as HbbTV testing solution that supports Selenium doesn't exist anymore. There are no references on the internet other than this presentation.

[5] `https://cucumber.io/`

[6] `https://git-scm.com/`

[7] `https://gitlab.fokus.fraunhofer.de/christian.bromann/mpat-webdriver-demo`

[8] The application to test here is the MPAT demo application of the Fraunhofer Fokus that is often to show off HbbTV demos at web media symposia. It can be found at `http://52.58.133.10/MPAT-core/web/mws/`

Page	Tests
Start	• should open correct page • should collapse menu
Dynamic Ad-Insertion	• should open correct page • should have a video element
DVB-T2 Broadcast Probing System	• should open correct page • should not play video from the beginning • should start the video • should play video

Tabelle 6.1: Test Scenarios for MPAT HbbTV App

The test basically goes through 3 different pages on the HbbTV app and checks if the app has properly changed the view and shows the right content. Next to checking whether or not the menu can collapse and expand when switching between menu and content back and forth, it tests if videos are automatically playing or if they play on click. When using the page object pattern[9] the test for the *DVB-T2 Broadcast Probing System* page can look as simple as:

```
1  import BPS from '../page_objects/bps.page'
2  import { expect } from 'chai'
3
4  describe('MPAT HbbTV App - DVB-T2 Broadcast Probing System', () => {
5      // ...
6      it('should start the video', () => {
7          browser.keys('right')
8          browser.keys('enter')
9          expect(BPS.video.isExisting()).to.be.equal(true)
10     })
11
12     it('should play video', () => {
13         expect(BPS.isVideoPlaying()).to.be.equal(true)
14     })
15 })
```

Listing 17: Example Test for the MPAT HbbTV Application

[9] https://martinfowler.com/bliki/PageObject.html

Listing 17 above shows a snippet of one test written in Node.JS. It uses WebdriverIO[10] as automation client. The page object contains all selectors and assertion helpers to abstract away the automation details in the test file. Listing 18 shows an excerpt of the corresponding page object.

```
import { Page } from './mpat.page'

class DynamicAdInsertionPage extends Page {
    // ...
    get video () {
        return $('.video-wrapper > object')
    }

    isVideoPlaying () {
        return browser.waitUntil(() => {
            return browser.execute(function () {
                var videoSelector = '.video-wrapper > object'
                var videoObject = document.querySelector(videoSelector)
                return videoObject.playState === 1
            }).value
        }, 10000, 'video never started')
    }
}

export default new DynamicAdInsertionPage()
export { DynamicAdInsertionPage as Page }
```

Listing 18: Page Object in WebdriverIO

To test if the video is playing the page object executes a JavaScript snippet directly in the HbbTV environment that checks if the video object is in the right playing state. If this doesn't happen within 10 seconds the command throw an assertion error and would cause the test to fail.

The testsuite that was setup was running on three different Smart TVs[11] in parallel[12]. It took about 24 seconds until all devices have finished the tests. Doing this test in a manual old fashioned way would have taken around 4-5 minutes. Interestingly the oldest TV model (the Samsung SmartTV) always took the longest time and even failed sometimes the test because the browser crashed and the TV had to reboot. Other than

[10] `http://webdriver.io/`

[11] An LG with WebOS2.0 and HbbTV 1.5, a Samsung with HbbTV 1.0 and a Philips on a Presto OS with HbbTV 1.5

[12] A video of this can be seen at `https://twitter.com/bromann/status/890199053467885578`

that the suite was running stable and no flakiness could be determined. Unfortunately the network to which all TVs where connected to was closed up so that a Jenkins integration with the in-house version control system was not possible. Therefor it was not possible to create pull request gatekeepers that would run functional tests for each single code change on a real Smart TV. However in a real world scenario building sophisticated pipelines with Jenkins allows a variety of possibilities. It starts on the test reporter level and ends on triggering Jenkins jobs when certain events happen, e.g. a pull request or a code push to the master branch happen.

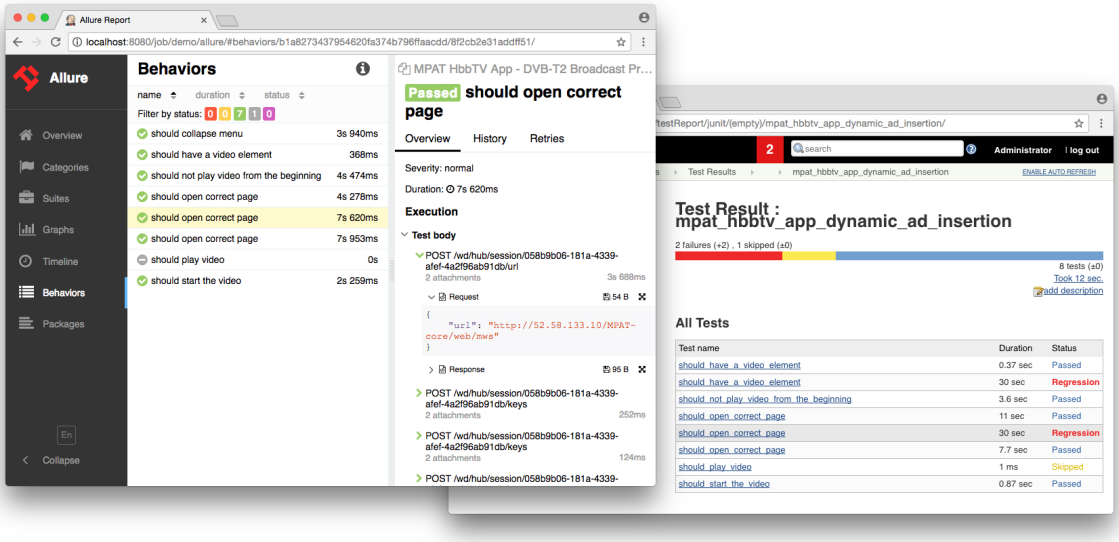

Fig. 6.4: Test Reports (left: Allure Reporter, right: JUnit Reporter)

Figure 6.4 shows two different test reports for the same test suite generated by WebdriverIO. These test reporters are like simple plugins and easy to set up. Other test frameworks in other languages can generate similar reports depending on the desires of the development team. With this solution such team can decided in which language to run their functional tests, on which Smart TVs supporting which HbbTV version and when to run these tests at what frequency.

6.2 Comparison To Other Testing Solutions

As of now there aren't any compatible testing solutions for HbbTV applications. The majority of developers still have to manually deploy their HbbTV applications on a server and setup a play out to access it on a real Smart TV device. However there is one company called Suitest that tries to simplify HbbTV testing with a click and play approach. It provides a service where developers can register their devices using a special hardware called Candy Box. The box is a *"control unit which Suitest uses to operate TVs, Set-Top-Boxes and other devices through their infrared port"*[Suitest, 2016].

In combination with a paid subscription this box can be used to connect arbitrary devices with the Suitest cloud. The company allows then to signup on their website to create tests and manage accounts. As described in section 2.1.4 the tests are being "clicked together" by a custom web interface where the developer can choose from a variety of action and assertion options. These tests can contain other tests which allows to split up multiple small tests to a big complex test suite. The concept of the service was patented in 2016 as an *"invention [that] relates to a method and a system for testing interactive applications on at least one TV device (18) according to at least one test scenario"*[KROCEK et al., 2016].

When signing up as a user at Suitest the first major difference immediately surfaces. The company provides their software as a service which means that it can only be used at higher scale by paying a monthly subscription. The test solution outlined in this thesis only requires hardware that costs once-only around 50€. The cheapest subscription plan at Suitest starts with 159€. There is also a free plan available to test the service. However this does not allow to run tests on a real Smart TV. In order to do that a Candy Box has to be purchased for 499€. Given that this box could instrument 40 devices the plan still doesn't allow to leverage the full potential of the Candy Box. Only with a paid subscription this box can be used properly.

The Candy Box itself is a useful piece of hardware that allows to instrument arbitrary TVs. Since it uses its infrared port it acts as remote control and has almost every ability to control the television like a human person. Connected to the internet and to the Suitest cloud every developer can control the TV using that box without having to be in the same location. The Raspberry on the contrary has certain limitations when it comes to controlling the TV. The instrumentation script of the DevTools Backend only works within a web environment. This means that the TV has to show an HbbTV app to allow the instrumentation script to work. This means that in order to run a test on the TV you need to turn it on and open an HbbTV application manually. Everything outside of the HbbTV context is not accessible. It is for instance not possible to open any internal menus or test a native application. These are indeed disadvantages against a box that uses an infrared connection to the TV. However looking at the scope of this thesis these limitations are acceptable. In addition to that the Raspberry Pi doesn't yet use any of the APIs that are provided by the manufacturer. Almost every TV can be accessed via an HTTP interface to gain control over certain features. These interfaces are used to allow mobile apps to work as e.g. a second remote control. Because the Pi is directly connected to the Smart TV, it has automatically access to these interfaces as well[13]. That being said, the way how the Candy Box can access the Smart TV can also be achieved using this approach.

Writing tests with the Suitest UI is fairly simple. As shown in figure 2.6 the UI provides an editor that allows to click together a chain of actions and assertions that can be read as normal English sentences. In order to choose elements from the app the developer

[13] As part of the research for this thesis a small Node.JS library (`https://gitlab.fokus.fraunhofer.de/christian.bromann/samsungtv`) was developed that can find and control newer Samsung Smart TV in the network by using its API interface.

can open up a utility menu that maps the mouse movements from the user to the app on the target device. Once the mouse hovers over an element it gets highlighted and can be chosen. In addition to that certain assertions, like the "has property" check, automatically fetch all available attributes the developer might be interested in. The editor in general feels slick and covers a variety of use cases that can ensure the expected state of the HbbTV application. Additional features like tagging or adding notes to certain steps are making tests not only easy to find but also easy to reason about. It doesn't require any technical experience to write, execute and maintain them. It is directed to not only app developers but also to project managers that want to make sure that certain requirements are met and features are implemented and stable.

The test environment developed with this thesis follows a completely different goal. It is based on already established industry standards like WebDriver and targets engineers directly. Tests are not "clicked together" using a GUI. They have to be written in a programming language and are usually created and maintained by the same developers that also build the HbbTV application. To interact with an element the selector has to be known. Since usually programmers are familiar with the elements on the page it is more difficult for other people, in or outside of the team, to write tests. However the DevTools Backend can give insights about the DOM structure of the application and allows to find these selectors. Still this approach requires some technical skills. It is in general though non opinionated and allows to create custom delivery pipelines that is suited for any requirements a development team can have. While tests on the Suitest platform can only be scheduled in certain time intervals or on exact times, using the Appium HbbTV Driver it is possible to run tests at any desired point in time, e.g. when a certain release is being made or if new code is proposed and about to be merged into the code base. This allows creating gatekeepers that prevent bad code that causes regression to be introduced.

To summarize these points, the Suitest solution is a legitimized approach for quality assurance of HbbTV applications. The SaaS company sells a prescribed solution that is quite expensive though for a developer without much budget. The solution provided in this theses is build on top of established industry standards with proven success in other software markets. It is much more affordable and easier to integrate in common software pipelines. In long term this approach has a higher likelihood of being adapted in HbbTV developer workflows than the one provided by Suitest.

Even though their patent describes a similar testing product to the one outlined in this thesis, it still doesn't violate their claims since all components are based on open standards and protocols. In fact their patent description explicitely talks about a *"test scenario [that] is run and monitored by a test driver system separate from the at least one TV device"* [KROCEK et al., 2016] which describes a system that has the only purpose to test an interactive application. The components developed in this thesis individually have a different purpose but can be setup in a way to achieve a similar goal. Important here is that the Appium HbbTV Driver can not only be used for testing, it is an instrument for automation in general. This is supported by the fact that the driver theoretically could also be used to automate any WebKit based desktop browser and not

only televisions. Therefor non of the components sole function is to test an interactive application. There is no functionality in neither of them that would provide a testing mechanism. With that it is up to the person who uses the components whether to test something or to just run a set of automated steps on the TV. The focus here is on automation and not testing. This is a major differentiation.

7 Conclusion

More and more digital devices are getting connected to the internet and run applications that are build with web technologies. The HbbTV protocol was the foundation of bringing the web on the big screen. This fairly new standard now encounters the same issues of debuggability and testability as other device and software markets before, e.g. in mobile application development. Until today there is no solution that really addresses these issues. As the HbbTV standard evolves and the device fragmentation increases the demand for sophisticated tools for quality assurance rises too. This thesis solves these issues by connecting established technologies and protocols together to create a testing and debugging solution that can be integrated in modern software development pipelines.

7.1 Summary

The work on this thesis started with an in-depth research phase on the HbbTV standard and current testing technologies. Unfortunately there were no existing solutions found at this point. Some companies like BBC provided some resources of their own homegrown solutions[1] that are suited for their requirements. However the goal of this work was to find a way to make testing and debugging of arbitrary HbbTV applications accessible to everyone. Therefor it has to be non opinionated and should not prescribe a certain way of how to do things. To evaluate the technical opportunities, one of the first steps was to discover the accessibility of a modern Smart TV. Using tools like WireShark a variety of interfaces could be discovered that allowed mobile applications to e.g. remote control the TV and enable second screen technology. These interfaces also gave insights on the device capabilities as well as their underlying software. Given that there were multiple manufactures with different Smart TV platforms that even have different interfaces based on their models, this approach of controlling a TV device turned out to not be a very promising strategy.

Another interesting view point were virtual machines that emulated an HbbTV environment on your operating system. With the Opera TV Developer tools[2] the company that made the Opera browser provides a set of tools and software that even allows developers to remote debug HbbTV applications from the browser. With the Opera TV Emulator it was not only possible to debug the app using the Chrome DevTools[3] it also allowed to run automated WebDriver tests by attaching the ChromeDriver to the Chro-

[1] See BBC Hive Open Source Test Tools (`http://bbc.github.io/hive-ci/`)
[2] `http://www.operasoftware.com/products/tv/tv-developer-tools`
[3] `https://wiki.operatv.tv/display/OTV/Debugging`

me DevTools Session that was run by the WebKit engine inside the emulator[4]. The idea of being able use the Chrome DevTools, which is the most powerful tool these days to inspect modern web apps, and test HbbTV applications using the WebDriver protocol, which drives millions of automated tests on mobile and desktop every day, seemed to be really compelling as a developer. However being restricted to an emulated environment is suboptimal and would at the end not solve the problem of quality assurance in a market of high device fragmentation like in the TV market.

The requirements of today's software delivery processes have been clearly influenced by the principles of agile software development. The demand to ship qualitative software faster also applies for the TV market where advertisement campaigns and broadcast content is getting rolled out constantly. To build a pipeline that provides such quick delivery cycles it requires *"an automated set of tools from code to delivery"*[Lehtonen et al., 2015]. The tools within the delivery process have to be highly interoperable to allow to build a pipeline that fits the developers needs since every software is different. Therefor the debugging and testing platform for HbbTV has to be interoperable too, which it only can be if it relies on open standards and protocols. Selenium, which is now standardized under W3C[5] as WebDriver, is such an open standard. It was developed as open source software since 2004. As industry leader for automated testing it runs millions of tests every day in desktop and mobile environments. To build a testing tool for HbbTV that is future proofed and aligns with today's software development standards it only makes sense to build on top of this technology. The WebDriver protocol itself only defines an interface of methods and properties to automate a browser. However it doesn't specify how the automation is being implemented. Therfore there are dedicated driver softwares for each individual browser. These drivers use different approaches to execute a certain WebDriver command on the application. Interestingly the driver for the Chrome browser uses a protocol that also drives one of the most powerful tools to develop and debug modern applications in Chrome. This protocol is called Chrome DevTools Protocol[6] which describes an interface to interact with the browser on multiple different domains like DOM, CSS or Network. It is build into WebKit which is a browser engine that drives a lot of other browser including Opera and Safari. The browser that render HbbTV applications on Smart TVs are also mainly driven by WebKit. Unfortunately it can't be accessed programmatically. The protocol still provides the ideal baseline for implementing a development and testing platform since it can be integrated in tools like Chrome DevTools.

That being said, the major programming effort for this thesis was to re-implement all interfaces that are described in the DevTools protocol as a standalone script. The use case model was the Chrome DevTools application. The goal was to create a script that once injected into a web environment can be used to connect to a Chrome DevTools application. To provide support for the Network domain a proxy was developed that captures network packages requested by the app. Obviously it is not possible to support

[4] `https://wiki.operatv.tv/display/OTV/Selenium+testing`
[5] `https://www.w3.org/`
[6] `https://chromedevtools.github.io/devtools-protocol/`

all methods described in the protocol using just the JavaScript API. In fact the protocol is very comprehensive, implementing all methods would be impossible time-wise. Therefor the scope was limited to support the most important tools, e.g. the Element, Console and Network tab. By inspecting the DevTools applications using the DevTools application itself and reverse engineer the communication between the application and a normal website it was possible to adopt the sequence of events and commands to a standalone script. This allowed an interesting in depth view of how modern HbbTV applications from private broadcasters like Pro7 or public broadcasters like ARD/ZDF were build. It showed that many channels are using the same application framework and only differ in design and content based on the brand.

The last part of the thesis was to build an automation driver to run WebDriver tests on Smart TVs running HbbTV applications. Appium already established a framework for building automation drivers based on the WebDriver protocol. It is used in drivers of different platforms like iOS, Android, Windows and Mac. Similar to how ChromeDriver and SafariDriver automates their browser, the HbbTV driver uses the implemented standalone script to execute WebDriver commands in the application.

7.2 Dissemination

The Fraunhofer Institute for Open Communication Systems will use this tool in near future to build and test their HbbTV applications on daily basis. Their TV laboratory will be equipped with a handful Raspberry Pis running the Appium HbbTV Driver and DevTools Backend components to debug and run tests on different TV models. This will be the foundation of a larger Selenium Grid that will allow to run tests on multiple devices at the same time.

In addition to that a talk with the title *"Appium for Couch Potatoes: an HbbTV Driver"* was proposed and accepted at the Selenium Conference 2017 in Berlin[7]. It will introduce the Appium HbbTV Driver component and will show off how to run automated tests on Smart TVs. Being able to test connected devices using Appium fits perfectly into the projects vision. Therefor the thesis was already mentioned in 2016 at the Selenium Conference in London in a talk by Jonathan Lipps with the title *"StarDriver Enterprise Appium to the Future"*[8]. Other abstracts and talk proposals had been sent out but haven't got accepted until due date.

7.3 Problems Encountered

During the implementation of the DevTools Backend component most of the problems were encountered due to the limitation of the HbbTV browsers and the restriction of not being able to debug code that is running on a Smart TV. Until today the functionality of the tool is limited for older TV generations that don't support the JavaScript APIs

[7]Talk proposal: `https://www.seleniumconf.de/talks#christian-bromann`
[8]Video available on YouTube: `https://youtu.be/e61OhZzbsEI?t=16m42s`

that have been used in the instrumentation script. These issues couldn't be solved due to the lack of time. Running a Selenium Grid without manual interaction is also currently still an issue. The Appium HbbTV Driver can only interact with a device when the instrumentation script was injected into an HbbTV app. In order to run tests each device has to be initially setup which includes starting the TV and open an HbbTV app. This has to be automated. Using the ethernet connection from the Raspberry this can be achieved with standards like Wake on LAN to boot up the TV or the rest interfaces provided by the Smart TV itself to simulate remote control events. Due to time limitation this was also skipped.

In addition to that some other interesting issues came up while developing the proxy server. The first version of it intercepted all packages which was a lot of overhead since the component was only interested in HbbTV documents in order to inject the instrumentation script. All assets only had to be tracked but not modified. In order to achieve that there has to be a filter that checks incoming requests to be forwarded or intercepted. The problem with that is that the request object itself, including its headers, doesn't always reveal information about the content type of the requested source. Also the filename can't be used to certainly determine the response. The solution for this problem was to make a HEAD request in advance. The response of this request contained the response headers which can be reliable used to determine the response type. Another issue that made some HbbTV applications fail to load were hostname changes. Some apps, e.g. the HbbTV app of RTL2, change the hostname between request and response. In this example the AIT package is referencing the HbbTV app at `www.rtl2.de/hbbtv`. Due to internal redirects the response contains a different host (`http://hbbtv.rtl2.de`). This affects some assets that don't contain a hostname in the URL. A JavaScript file referenced with `/js/app.js` now gets loaded from `www.rtl2.de/js/app.js` instead of `http://hbbtv.rtl2.de/js/app.js`. As result the HbbTV application can't be initialized successfully because some important assets were missing. To workaround this issue the proxy has to indentify all script and link tag references and has to make sure that they contain an absolute URI based on the response host name. This works fairly well but is definitely not the ultimate solution for this problem.

7.4 Outlook

The current developed prototype is more or less just a proof of concept of how modern debugging and testing solutions can be applied to the TV screen. There are numerous enhancements that can be made. So far debugging and testing is limted to the HbbTV space only. However there are also native applications that need do be tested. A tighter integration into the developing frameworks of each TV platform would allow to provide some sort of test automation also for native applications[9]. These integration are mainly hidden behind the rest API interface of the Smart TV. Unfortunately they are not well

[9] For example the TV platform Tizen OS which is deployed on Samsung devices has an interface for native applications that is also based on Chromes DevTools Protocol (`http://developer.samsung.com/tv/develop/getting-started/using-sdk/web-inspector/`).

documented and have to be discovered for each individual platform.

Another really interesting opportunity would be to offer this software as a service and allow anyone access to Smart TV devices. Similar to the SaaS model of other cloud testing companies like SauceLabs[10] or Perfecto Mobile[11] this service would allow customers to signup and access real Smart TV devices based on their subscription plan from anywhere in the world. Broadcast companies would not need to setup their own TV device lab anymore but instead just use this service. Next to remote debugging and automated testing features it could offer some sort of manual testing where the customers can choose a TV in the device lab and connect to it via a dedicated A/V channel. This can be achieved by mirroring the A/V stream of the TV to a server that can transform this into a video stream. A simpler option would be to just record the screen with a normal camera. Given that the number of countries that support HbbTV is growing and more broadcast companies are going to be interested in this, a company that would offer such service is very likely to be successful. Especially since the HbbTV technology is new and many new features haven't been implemented in TVs yet the device fragmentation will be growing and increasing the demand for automated testing.

[10] `http://saucelabs.com/`
[11] `https://www.perfectomobile.com/`

List of Figures

List of Listings

List of Tables

List of Acronyms

AIT	Application Information Table
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ARM	Advanced RISC Machine
A/V	Audiovisual
CE	Conformité Européenne
CI/CD	Continuous Integration and Delivery
CSS	Cascading Stylesheets
DASH	Dynamic Adaptive Streaming over HTTP
DevOps	ßoftware DEVelopmentänd ïnformation technology OPerationS”
DHCP	Dynamic Host Configuration Protocol
DHTML	Dynamic HTML
DOM	Document Object Model
DVB	Digital Video Broadcasting
e2e	end-to-end
ETSI	European Telecommunications Standards Institute
FOKUS	Fraunhofer Institut fuer offene Kommunikationssysteme
GHz	Gigahertz
GUI	Graphical User Interface
HbbTV	Hybrid Broadcast Broadband Television
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
MHP	Multimedia Home Platform
MPEG	Moving Picture Experts Group
QA	Quality Assurance
REST	Representational state transfer
SaaS	Software as a Service
SSH	Secure Shell
SSL	Secure Sockets Layer
UPnP	Universal Plug and Play
USB	Universal Serial Bus
uuid	Universally unique identifier
W3C	World Wide Web Consortium
WIPO	World Intellectual Property Organization

Bibliography

[Barron, 2015] Barron, B. (2015). Why you should start using chrome developer tools right now. last checked on 08/08/2017. https://www.elegantthemes.com/blog/resources/why-you-should-start-using-chrome-developer-tools-right-now.

[BEPEC, 2017] BEPEC (2017). What exactly is devops? last checked on 08/08/2017. https://www.bepec.in/single-post/2017/02/09/What-exactly-is-DevOps.

[Berg, 2012] Berg, A. (2012). *Jenkins Continuous Integration Cookbook*. PACKT Publishing.

[Campbell, 2015a] Campbell, B. (2015a). Hbbtv app testing - challenges and solutions. Technical report, Eurofins Digital Testing. last checked on 08/08/2017. http://dtg.org.uk/publications/HbbTV/35_BobCampbell_HbbTV_Presentation.pdf.

[Campbell, 2015b] Campbell, D. B. (2015b). Whitepaper on hbbtv testing for broadcasters and operators. Technical report. last checked on 08/08/2017. https://goo.gl/wwtBz6.

[Cohn, 2009a] Cohn, M. (2009a). The forgotten layer of the test automation pyramid. *Mike Cohn's Blog–Succeeding with Agile*. last checked on 08/08/2017. https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

[Cohn, 2009b] Cohn, M. (2009b). *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 1st edition.

[CrossBrowserTesting, 2017] CrossBrowserTesting (2017). Moving from manual to automated testing - a tester's journey. last checked on 08/08/2017. https://crossbrowsertesting.com/pdfs/moving-from-manual-to-automated-testing.pdf.

[Developer, 2015] Developer, H. (2015). Setting up a development environment — hbbtv developer,. last checked on 08/08/2017. http://www.hbbtv-developer.com/site/wiki/index.php?title=Setting_up_a_development_environment&oldid=658.

[ETSI, 2012] ETSI, T. (2012). 102 796 v1. 2.1 (2012-11). *Hybrid Broadcast Broadband TV*. last checked on 08/08/2017.

[eurofins, 2017] eurofins, D. T. (2017). Testwizard tool suite. last checked on 08/08/2017. http://www.eurofins-digitaltesting.com/media/28855/testwizard.pdf.

[Fain et al., 2014] Fain, Y., Rasputnis, V., Tartakovsky, A., and Gamov, V. (2014). *Enterprise Web Development: Building HTML5 Applications: From Desktop to Mobile*. O'Reilly Media.

[Gartnet, 2017] Gartnet (2017). Gartner it glossary - addressable tv advertising. last checked on 08/08/2017. `http://www.gartner.com/it-glossary/addressable-tv-advertising`.

[Google, 2017a] Google (2017a). Chrome devtools protocol viewer - latest (tip-of-tree). last checked on 08/08/2017. `https://chromedevtools.github.io/devtools-protocol/`.

[Google, 2017b] Google (2017b). *Chrome DevTools Protocol Viewer - latest (tip-of-tree) - DOM Domain*. Google. last checked on 08/08/2017. `https://chromedevtools.github.io/devtools-protocol/tot/DOM/#event-documentUpdated`.

[Google, 2017c] Google (2017c). Ten things we know to be true. last checked on 08/12/2017. `https://www.google.com/about/philosophy.html`.

[KROCEK et al., 2016] KROCEK, L., PEREVORSKYY, T., and NEDELJKOVIC, M. (2016). Method and system for automating the process of testing of software application. WO Patent App. PCT/EP2016/060,586. `http://www.google.it/patents/WO2016180894A1?cl=en`.

[Langel, 2017] Langel, T. (2017). Testing the open web platform. last checked on 08/08/2017. `https://www.w3.org/blog/2013/02/testing-the-open-web-platform/`.

[Lehtonen et al., 2015] Lehtonen, T., Suonsyrjä, S., Kilamo, T., and Mikkonen, T. (2015). Defining metrics for continuous delivery and deployment pipeline. In *SPLST*, pages 16–30.

[Maksimović et al., 2014] Maksimović, M., Vujović, V., Davidović, N., Milošević, V., and Perišić, B. (2014). Raspberry pi as internet of things hardware: Performances and constraints. *design issues*, 3:8. last checked on 08/08/2017. `https://www.researchgate.net/publication/272175660_Raspberry_Pi_as_Internet_of_Things_hardware_Performances_and_Constraints`.

[Merkel, 2010] Merkel, K. (2010). Hbbtv — a hybrid broadcast-broadband system for the living room. *EBU technical review-2010 Q*, 1.

[Podhradsky, 2013] Podhradsky, P. (2013). Evolution trends in hybrid broadcast broadband tv. In *ELMAR, 2013 55th International Symposium*, pages 7–10. IEEE.

[Suitest, 2016] Suitest (2016). Candy box. last checked on 08/08/2017. `https://suite.st/docs/devices/candybox/`.

[Syed, 2017] Syed, B. A. (2017). *TypeScript Deep Dive*. Samurai Media Limited. last checked on 08/08/2017. `https://www.gitbook.com/book/basarat/typescript/details`.

[Xu et al., 2016] Xu, Y., Xie, S., Chen, H., Yang, L., and Sun, J. (2016). Dash and mmt and their applications in atsc 3.0. *ZTE COMMUNICATIONS*, 14(1):39–50.

Annex

```
In Regular Operation: Australia, Austria, Bosnia and Herzegovina, Bulgaria,
Czech Republic, Denmark, Estonia, Finland, France, Germany, Hungary, Italy,
Luxembourg, Madagascar, Malaysia, Mauritius, Namibia, Netherlands, New Zealand,
Norway, Poland, Saudi Arabia, Senegal, Singapore, Slovakia, Slovenia, Spain,
Sweden, Switzerland, Turkey, United Arab Emirates, United Kingdom

Announced / Trials: Croatia, Gambia, Ireland, Ivory Coast, Jordan, South Africa

Under Consideration: Algeria, Belgium, Brazil, Burkina Faso, Chad, China, Columbia,
Egypt, Ghana, Guinea, Indonesia, Iran, Iraq, Liberia, Libya, Mali, Mauritania,
Morocco, Myanmar, Niger, Oman, Portugal, Russia, Sierra Leone, Sudan, Syria,
Thailand, Tunisia, Vietnam, Yemen
```

Listing 1: A list of countries where HbbTV is - will be - or is considered to be deployed

```
st:redpro7
cp:p7de_Homepage
sv:ke
co:http://hbbtv.prosieben.de/Startseite/P7 Comedy Male Dashboard_15089/The
   Big Bang Theory_1389783/others/others
pt:CP
rf:hbbtv.redbutton.de
r2:http://hbbtv.redbutton.de/service/redbutton.php?brand=p7de
ur:hbbtv.prosieben.de
xy:1920x1080x24
lo:DE/Berlin
cb:000d
vr:312
id:jvaob5
lt:1499955480751
ev:
cs:oj7bdp
mo:1
```

Listing 2: Request to INFOnline (http://de.ioam.de) containing user data

```
a:24863135
sa:1
v:974.7d740e1
t:Unnamed Transaction
rst:2326
ref:http://hbbtv.prosieben.de/v2/dashboard/p7de
be:-509
fe:2555
dc:1368
af:err,xhr,ins
perf:{"timing":{"of":1499955479727,"n":0,"f":0,"dn":0,"dne":0,"c":0,"ce":0,
      "rq":0,"rp":0,"rpe":347,"dl":314,"di":1859,"ds":1859,"de":1995,"dc":3020,
      "l":3046,"le":3062},"navigation":{}}
ja:{"engineName":"WebKit","deviceType":"HbbTV","userAgentOS":"Samsung",
    "hbbtvVersion":"1.2.1", "hbbtvVendorName":"Samsung",
    "hbbtvModelName":"SmartTV2016","hbbtvHardwareVersion":"T-HKMFKDEUC",
    "hbbtvSoftwareVersion":"1160.6"}
jsonp:NREUM.setToken
```

Listing 3: Request parameter to New Relic aggregator (http://bam.nr-data.net)

```
p:435,http://hbbtv.prosieben.de/Menue-Seiten/Magazin/Home/home/#100,1,
  1920x1080,24,1,1499943844946,2,1280x720,1
tz:2
cet:120
ct:Rubrik_Klick
la:de
pu:http://hbbtv.prosieben.de/v2/magazine/p7de
ck1:p7de
ck2:Magazin
ck3:others
ck10:Rubrik_Klick
ck11:Stars
ck12:col_1-row_2
ck14:Sub-Homepage
ck19:http://hbbtv.prosieben.de/v2/magazine/p7de/400
ck23:not_set
ck24:not_set
ck25:364_36
ck26:section-link
ck27:74_191
eor:1
```

Listing 4: Request parameter to internal Pro7 server (`http://hbbtv-track.redbutton.de`)