Diagnosing User-Visible Performance Problems in Production High-Density Wi-Fi Networks

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Nathan D. Mickulicz

B.S. in Electrical and Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May, 2018

Dedicated to my mother, Linda,

for going on this journey with me and supporting me every step of the way.

Acknowledgements

Pursuing a Ph.D. has been a rewarding experience, one that has given me many opportunities for both academic and personal growth. However, I would have not been able to complete my thesis without the help, support, and guidance from many people along the way. I would like to thank all of those who have been a part of this journey with me.

First and foremost, I would like to thank my parents, Donald and Linda Mickulicz. Throughout my life they have supported me, believed in me, and encouraged me to be all that I can be. My mother has been there for me through all of the ups and downs along this journey, and without her support I may not have made it to the end. Throughout my childhood, my father always pushed me to stretch myself and do better. I regret that he passed away before he could see me finish my Ph.D., but I know he would have been proud.

I would like to thank my advisor, Prof. Priya Narasimhan, for giving me this opportunity to earn my Ph.D. and for everything that she has done to guide me in developing my thesis. I have worked with Priya since I was an undergraduate student, and throughout my undergraduate and graduate studies she has taught me how to develop research ideas, write about them scientifically, and present them to others. Whenever I have needed help or advice, she has always been there for me. I can't thank her enough for being my advisor, my mentor, and my friend.

I would also like to thank the members of my thesis committee, Dr. Rajeev Gandhi, Prof. David O'Hallaron, and Prof. Dan Siewiorek, for their feedback that provided direction for my thesis work. Dr. Rajeev Gandhi advised me regarding problem-diagnosis algorithms and provided guidance on developing the algorithms that I used in my approach. Prof. David O'Hallaron suggested comparing the Wi-Fi experiences of multiple clients to diagnose problems, which gave me inspiration for my diagnosis and mitigation approaches. Prof. Dan Siewiorek advised me regarding approaches to anomaly detection, and I want to thank him for his fault-injection expertise that helped to guide my experimental work. I deeply appreciate the time that each one of them took to review my work and provide me with thoughtful and insightful feedback.

I am grateful for the help and support of other researchers who have contributed to my work and provided examples of their own research work for me to follow. During my time at CMU, I've been fortunate to work with Utsav Drolia, Mike Kasick, Rolando Martins, Soila Pertet, Joe Slember, and Jiaqi Tan, and I want to thank each one of them for helping to guide my work. I want to give special thanks to Utsav Drolia and Rolando Martins for collaborating with me on my thesis work and for their contributions to my research papers.

I would like to thank Prof. Greg Ganger and the Parallel Data Laboratory community at CMU for giving me many opportunities to present my work to industry at PDL events. The feedback I received from those presentations was invaluable in shaping my work.

I would also like to thank the Pittsburgh Penguins for providing me with the opportunity to test my approach at the UPMC Lemieux Sports Complex in Cranberry, PA. The data that I gathered from my experiments in their facility was very helpful in validating my approach.

During my dissertation research, I received financial support from the Intel Science and Technology Center for Cloud Computing (ISTC-CC) and Embedded Computing (ISTC-EC), Carnegie Mellon's CyLab and Parallel Data Lab, and from the CMU-Portugal Program. I also received support in the form of equipment and data from YinzCam, Inc.

Abstract

Large-scale, high-density Wi-Fi networks use hundreds of access points to serve thousands of closely-packed users within a large physical space (hundreds of thousands of square feet or more, such as in a stadium or arena). Because of their scale, these are complex and dynamic systems comprised of several layers and multiple components within each layer, and faults may be present in any one of these components. The problems that manifest from these faults are usually not network-wide and may be localized to a certain physical areas of the network. This makes these problems challenging to detect and diagnose; in most cases, only a small number of devices tend to be impacted by any given problem. However, many such problems may occur simultaneously in different areas of the network. Adding to the complexity is the dynamic nature of such networks, where the physical positions of radios (in end-user devices), human bodies, and other objects in the space are constantly changing, thereby creating a continually-changing RF environment. Taken together, these properties make problem diagnosis in large-scale, high-density Wi-Fi networks challenging. There are many existing techniques for diagnosing problems in Wi-Fi networks. Many of these approaches rely on data from only a single perspective of the network to diagnose problems, for example, either the client, the infrastructure (access points), or external Wi-Fi sensors that passively monitor the network. In addition, many of these approaches require the invasive modification of the network's components in order to collect data, through techniques such as the installation of specialized software on clients, modifying the firmware on access points, or even physically installing specialized devices in the RF environment of the Wi-Fi network. Finally, many approaches rely on offline analysis of the collected instrumentation, in which case diagnosis cannot be done in real time (minutes or less). Many others require network connectivity for real-time diagnosis, in which case the device must be able to communicate using the Wi-Fi infrastructure (that may be experiencing a problem). As a result, many of these approaches are difficult to deploy in production networks (due to the high financial cost or maintenance effort required), and those that are deployed often fail to detect and diagnose problems that are localized to a small number of devices (10 or less) or problems that are only present for a short time (minutes or less).

This dissertation takes a unique approach that contrasts with existing approaches in three key ways. First, we combine the Wi-Fi performance data from multiple layers of the Wi-Fi network and attempt to diagnose problems at all of these layers, rather than focusing on a single layer alone, and we introduce a fault model that includes faults that can occur across all layers of the system. Second, we require no invasive modification of the Wi-Fi network or its components in order collect data and perform problem diagnosis and mitigation. Third, we present an infrastructure-free approach to problem diagnosis that

relies on Bluetooth communication with other devices nearby (peers) to perform diagnosis based on multiple perspectives of the Wi-Fi network. With this approach, our diagnosis algorithm is able to collect data from multiple network perspectives without relying on Wi-Fi infrastructure, which may be slow or unavailable. Our approach begins with the construction of an instrumentation and data-collection system to obtain Wi-Fi performance metrics from both the client and infrastructure perspectives of the network. We then build upon our instrumentation to determine when user-visible problems occur. We define a user-visible problem as a Wi-Fi-network-performance problem that causes users to disengage from using the network. Once we have detected a user-visible problem, we then proceed to diagnose the root cause of the problem as one of the faults in our fault model using an approach based on decision trees. Finally, based on the diagnosed fault, we apply an automated mitigation-strategy, which forces the device to associate with a different access point that will likely provide better performance.

To validate our approach and demonstrate its real-world impact, we have conducted a number of studies to collect data in support of our approach from both a laboratory testbed and real-world production Wi-Fi networks. We used our instrumentation and data-collection system to obtain data from over 25 real-world, large-scale, high-density Wi-Fi networks located within collegiate and professional stadiums. Our diagnostic system was deployed in a real-world mobile video-streaming application used over the Wi-Fi networks in these stadiums. Using this data, we determined the thresholds for when a Wi-Fi performance problem becomes user visible, based on our study of when users disengage from using the video-streaming application in the face of buffering. In addition to obtaining real-world data, we have studied this phenomenon in a testbed for fault injection and diagnosis that has been deployed both in a lab environment and in an arena to collect data on the behavior of large-scale, high-density Wi-Fi networks and understand how best to diagnose problems. Using this testbed, we evaluated the performance of our problem-diagnosis approach in terms of its precision and recall on injected faults. We also evaluated the performance of our mitigation strategy on our testbed by injecting faults and verifying that the selected mitigation strategy successfully mitigated the problem caused by that fault. We found that our approach diagnoses the correct root cause of faults with high precision and recall (often above 90%) and can mitigate problems via alternative access-point selection in 100% of our test cases. While we have studied our approach in certain test environments and for video-streaming applications, we believe that our approach can be applied to any Wi-Fi network and many other applications outside of video streaming. Our work in this dissertation could be extended through the automated discovery of the parameters for our diagnosis and mitigation algorithms that provide the best performance in other Wi-Fi networks, along with further studies of how Wi-Fi performance problems manifest in other types of applications and under what conditions users disengage with those applications due to problems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large-scale, high-density Wi-Fi networks are those Wi-Fi networks that use hundreds of access points to simultaneously provide network connectivity to tens of thousands of simultaneous users, covering physical spaces measuring in the hundreds of thousands (arenas) to millions of square feet (stadiums, airports). These networks are large-scale because of the number of clients that they serve relative to smaller-scale Wi-Fi networks commonly found in homes, offices, and small public spaces where there are typically less than 100 users. These networks are also high-density because users are often packed close together (for example, in a seating section of a stadium) with each access point serving tens or hundreds of clients simultaneously. Today, these large-scale, high-density Wi-Fi networks are commonplace in our daily lives. We rely on these networks keep us connected to each other as we move throughout our homes, offices, schools, hospitals, airports, and even entertainment venues [54]. These Wi-Fi networks must support all of the demanding low-latency/high-bandwidth applications that are pervasive on our mobile devices, such as live-video streaming, photo sharing, and even just browsing websites and mobile applications that are rich in multimedia content. Indeed, we now have an expectation that these networks will be available for our use whenever we are in a public space, and as a result, we have come to depend upon them in order to conduct our daily lives. As a result, it is important that these networks be dependable; the automated diagnosis of Wi-Fi performance problems is a step towards constructing a dependable Wi-Fi network.

The complex and dynamic nature of large-scale, high-density Wi-Fi networks poses a number of challenges to detecting, diagnosing, and mitigating the performance problems that can occur within them. Faults in Wi-Fi networks rarely cause a global outage throughout the entire network; the backend systems and infrastructure that operate the network are engineered for robustness and high availability, typically being configured in fail-over pairs with extensive monitoring. Thus, the problems that impact end-users are almost always local to a physical area of the network, typically within the range of one or two access

points (APs). Indeed, almost every Wi-Fi user knows that if the network performance is poor in one location, moving to another location frequently resolves the problem. Because these are complex systems involving hundreds of access points and thousands of end-user devices that are constantly entering and leaving the system, it can be difficult to pinpoint where a problem is occurring, especially from the infrastructure perspective. Users that are experiencing problems do not typically leave their device connected for long. After a minute or two of being unable to use the network, users will either switch to an alternate network (e.g., cellular) or give up entirely, which affords only a short amount of time to detect and diagnose the problem. In addition, these networks are dynamic and constantly changing, as devices and their users move around the physical space, resulting in a RF environment that changes with time. Furthermore, the applications in use on each end-user's device is constantly changing as well, as users switch between social networking, web browsing, video streaming, and other applications, thus creating a constantly-changing workload. This means that Wi-Fi problems are often transient, and can come and go over time as workloads change and people move throughout the physical space.

Through our study of mobile video-streaming applications operating in real-world high-density production Wi-Fi networks, we have observed the types of faults that occur in these networks, how they manifest as errors on user devices, and how real-world system operators attempt to mitigate these faults in production environments. The network operators provision a such a Wi-Fi network with resources and capacity intentionally at "idle" times when there is minimal user-demand on the network, using a combination of site-survey tools and off-the-shelf bandwidth tests. During times of peak user demand, operators use infrastructure-side monitoring to collect data about network performance and potential problems. Because all monitoring is done from the infrastructure side, some problems that cause user-visible errors, such as the inability to connect to the wireless network, are not visible to operators. For those problems that are visible to operators, there are few options for immediate mitigation, as changing the infrastructure configuration during peak user-demand carries with it a high risk of introducing additional faults into the system, through either system malfunctions or human error. Instead, errors are usually logged, and mitigation (such as changing AP locations or adjusting signal power) is applied during a subsequent period of low user-demand.

In addition to the state of practice in industry, we have also studied contemporary academic approaches to Wi-Fi problem diagnosis. These approaches use various techniques for data collection, problem diagnosis, and mitigation, and are all successful, to some degree, in mitigating Wi-Fi performance problems. In aggregate, these related approaches cover the full spectrum of possible Wi-Fi performance problems. However, all of these approaches fall short in at least one of three key areas: (i) the range of faults covered by the fault model, (ii) the financial cost or amount of human effort involved in deploying

and maintaining the system, or (iii) the time-delay between the incidence of the problem and determining the diagnosis of that problem. Many modern techniques focus on diagnosing problems at a single layer of the network, for example, at the wireless-link layer alone [48, 59]. These approaches are insufficient to diagnose the full range of problems that can occur in a Wi-Fi network, as problems can arise in all of the layers of the system (i.e., end-user devices, wireless links, access points, backend/wired infrastructure). Most approaches come at a significant, often impractically-high, cost to implement in a production system. For example, the series of results by Cheng et. al. [18, 16, 17] rely on air monitors, which are specialized hardware deployed throughout the authors' RF environment to capture packets transmitted from all client devices. These are cost-prohibitive to deploy and maintain at scale. Furthermore, many other techniques rely on invasive modification to clients [10], access-points [77], or both [55], strategies that are not feasible in production environments where users bring their own client devices and system administrators tend to resist modifications to functional infrastructure equipment. Finally, many approaches to the problem take an offline or non-real-time approach to diagnosis, while studies [52] have shown that users have attention-spans measured in seconds, and tend to quickly disengage or switch to alternative networks if the Wi-Fi network is not working properly. For example, Shaman [17], WiFiProfiler [14], and WiFiSeer [73] all require users to interact with a separate graphical user-interface to report Wi-Fi problems and receive a diagnosis, rather than that diagnosis being automatically triggered by problem detection.

Effectively diagnosing and mitigating performance problems in production Wi-Fi networks requires the confluence of a fault model that includes faults across all layers of the system, an implementation that is inexpensive to deploy and does not require significant human effort to maintain, and a diagnosis approach that can return a diagnosis within minutes of a problem being detected. To provide coverage of such a broad fault model, the system must detect and diagnose faults at multiple layers of the system. It is impossible to diagnose the root cause of all of the faults in the system without considering multiple layers. To avoid being too cumbersome or too costly to implement, diagnosis approaches must avoid the invasive modification of devices within the network, or the introduction of devices like air monitors that are too costly (and, therefore, impractical) to purchase and maintain. Finally, to provide near-real-time diagnosis, the approach must be able to collect the data required for diagnosis in real-time, either through local measurement or through communication with other devices in the network.

1.1 Provisioning Large-Scale, High-Density Wi-Fi Networks

Provisioning a large-scale, high-density Wi-Fi network is a challenging problem. Given a large physical space, such as a stadium or arena (hundreds of thousands of square feet), the Wi-Fi architects must decide

Figure 1.1: The YinzCam mobile application in action at a Pittsburgh Penguins NHL game.

the number of access points required, where those access points should be located, and how they should be configured in terms of several parameters such as signal power and channel assignment. While some of this is automated to a degree (e.g. automated channel assignment to avoid co-channel interference), often the RF environment in these spaces is complex enough that even those parameters may need to be manually set. Configuring such a Wi-Fi network for optimal performance throughout the space is a process of trial and error, as network architects change settings, test the results of those changes, and then test again, and so on.

Provisioning a Wi-Fi network for optimal performance is critical because of the applications that run on those networks. One such application is live video streaming. Today, via the Internet, anyone with a smartphone can subscribe to services providing live-video streaming from a variety of sources (television broadcasts, video on demand, etc.) and receive these streams via any Wi-Fi network providing Internet access. One specific example of this type of service is the streaming of live camera angles of a sporting event from within a stadium [53]. Using a Wi-Fi-enabled smartphone, a fan attending a game at the stadium can enjoy multiple live camera angles and replays from their seat, in the concourse, or wherever they are located in the venue. Figure 1.1 shows an example of streaming on three different smartphones

at a Pittsburgh Penguins NHL game in 2009. Fans using this service expect that this live-video-streaming application will be available everywhere in the building, and in order to meet that expectation, the Wi-Fi network must be capable of supporting video streaming at any place a fan may visit. Thus, this application and environment represent the challenges one must face in order to successfully provide Wi-Fi service in a large-scale, high-density environment.

1.1.1 Challenges

User density and scale. At sporting events, live-video streaming is used simultaneously by thousands of fans at each game. Furthermore, users are packed closely together whether in their seats watching the game or standing in line at a concessions stand.

Roaming. Fans at sporting events expect to be able to use live-video streaming whether they are sitting in their seat, walking around the concourse, or standing in line. This means that user density patterns and the RF environment are constantly changing, both during a game and one game-day to the next. It is unreasonable to expect that a one-time infrastructure configuration will perform well across all games at all times.

Interactivity. The live-video streams offered at a sporting event may be of the sporting event itself (for example, to provide unique perspectives of the event that fans could not otherwise get from their seat). Users expect to be able to keep up with the action even while they are not in their seats, and the user can easily verify if the video is really live. This means that the service needs to provide video from the in-venue source to smartphones with low end-to-end latency.

Bandwidth. Each live-video stream is transmitted at up to 2 Mbps average bit-rate (for high-definition video). Furthermore, due to technical challenges with multicast on smartphone platforms (and the latency that must be introduced to overcome those challenges), these streams are unicast to each smartphone playing a stream. Thus, each user watching live video consumes a unique 2 Mbps video stream.

1.1.2 Inconsistent User Experiences

Figure 1.2 shows the a typical arena seating chart showing a number of representative access points and users with smartphones. During a typical game, such as the one shown, the user experience varies widely across different locations in the arena. For example, the user in section 201 (bottom) reports an excellent experience with the live-video stream, while just across the arena the user in section 112 (top) is seeing poor visual quality stream (distortions and pixelation), and the user in section 213 (top-left) can't even connect to the Wi-Fi network. Furthermore, a user's experience can vary over the course of the game, even

Figure 1.2: A typical game-day environment with user feedback. Users are spread unevenly throughout the physical space. Some users experience performance problems, while others have good performance.

if that user is stationary. For example, the user above section 201 may report that he can no longer stream video later on during the game, while the user in section 112 will no longer experience problems. During the course of a game, a user's experience with live-video streaming can vary from poor to excellent over time as well as space.

1.1.3 Site Surveys

In order to provision a Wi-Fi network for a particular application, wireless vendors perform a site survey [49]. Figure 1.3 presents an example of a typical site surveying procedure. Technicians from the wireless vendor load the target application onto a small number of wireless devices (in this example, three smartphones) and move to a section of the arena (here, section 116). Another wireless technician connects to nearby access points and tweaks settings until the application performs at or above some minimum

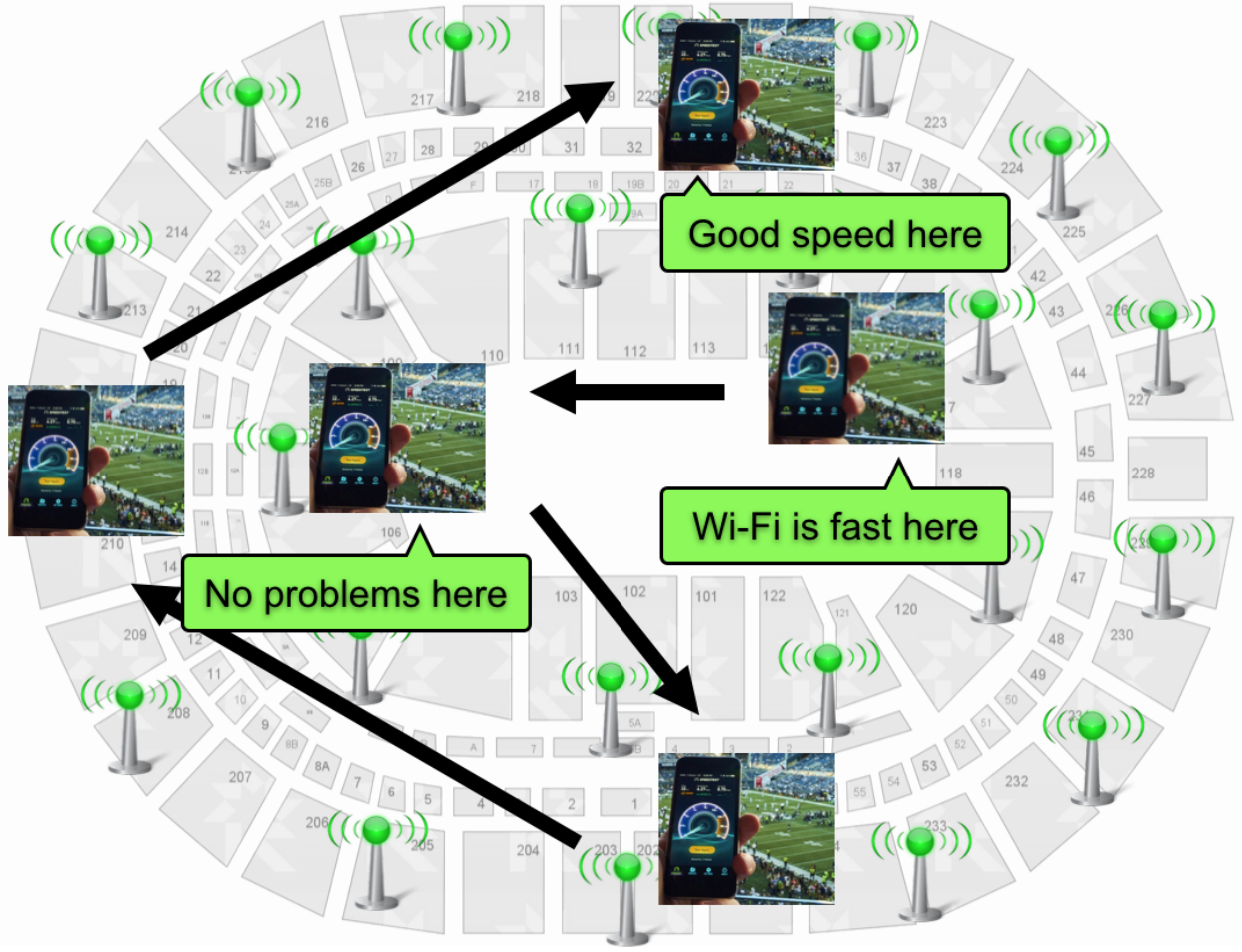

Figure 1.3: A typical site-survey procedure for a Wi-Fi network in a stadium. Wi-Fi engineers walk around the building while sampling Wi-Fi performance at various locations. The tools used often include Internet-speed-test mobile applications.

quality-of-service level set by the application vendor. Then, the technicians move on to another section of the arena (such as section 110) and repeat the procedure until the entire venue has been covered. There are many steps to a site-survey procedure, as outlined in [32], and many factors of the deployment to take into account when conducting a site survey and interpreting the results of the survey [12].

There are a number of reasons why this provisioning method does not produce good results in our target deployments. First, the transmission ranges of access points overlap, so changing the environment in one area may affect the provisioning in adjacent areas. In a site survey, this sometimes requires the re-testing of an area once adjacent areas have been tested and tuned. In the end, these sequences of local optimization may never approach the global optimum. Furthermore, the number of users in the physical space and number of objects in the RF environment are much lower than what is seen on game day. Even though an area might be provisioned to give all devices good signal strength to a nearby

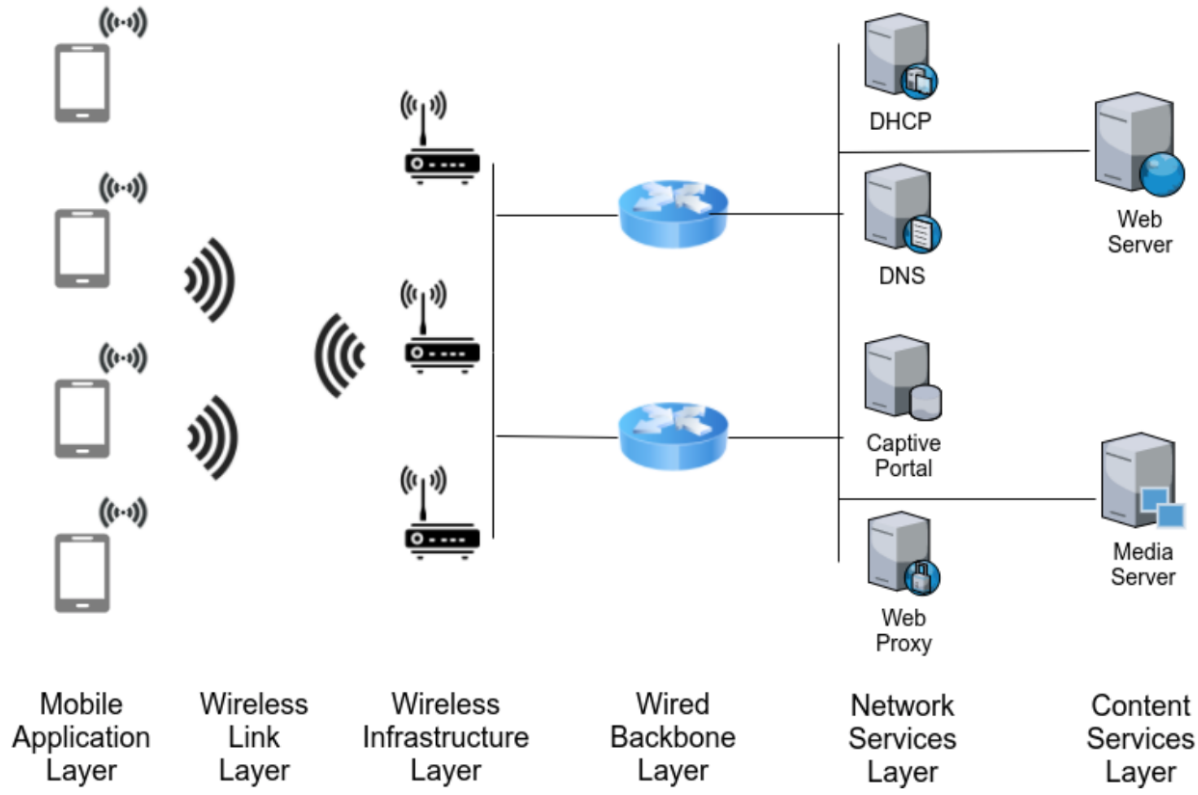

Figure 1.4: The layers of components in a typical large-scale Wi-Fi network.

AP, contention for the shared wireless medium and bandwidth limitations may still render the service unusable. Finally, usage and density patterns are constantly changing, both game-to-game and during a single game. Configuring corporate Wi-Fi networks to handle non-uniform user density has been studied extensively [8, 9]. However, unlike laptop-heavy corporate environments, users with smartphones are far more mobile in our environment [40]. Furthermore, mobile users can lower the throughput of an entire network if mobile devices are connected to far-away access points and compensate by transmitting at a lower bit rate [43]. For these reason, we believe that a single network-provisioning exercise on a non-event-day with a small number of devices is not flexible enough to provide good service under all conditions encountered in a production environment.

1.2 Wi-Fi Network Components and Architecture

This section describes the components that are required for the end-to-end operation of an application running on top of a Wi-Fi network. We divide these components into layers, with each layer containing those components that are related by the function that they perform (see Figure 1.4). Each of these components are a possible point of failure.

Mobile-Application Layer. The components in the mobile-application layer are mobile Wi-Fi clients, such as smartphones and tablets. This tier includes the physical hardware of the mobile device, the operating system and drivers and their configurations, and the application software that runs on the device and interacts with the user. Protocols commonly used at this layer are application-layer protocols, such as HTTP Live Streaming (HLS) for video streaming [6]. The most common type of fault that occurs at this layer is misconfiguration, such as disabling the Wi-Fi stack.

Wireless-Link Layer. This layer encompasses the RF environment and the physical space that wireless devices occupy. Instead of hardware or software systems, this layer contains the physical system that describes the propagation of Wi-Fi signals. Protocols used at this layer include the 802.11 link-layer protocol. The most common faults in this layer are channel overload due to a large number of Wi-Fi clients operating simultaneously on the same channel, noise from non-802.11 devices such as microwave ovens, and coverage gaps where signals are not able to reach some parts of the physical space due to physical obstructions (such as walls) or inverse-square-law signal degradation.

Wireless-Infrastructure Layer. This layer includes the the infrastructure devices that maintain connections with Wi-Fi clients in the mobile-application layer. Access points function as a network bridge between Wi-Fi clients and the wired network backbone. In addition, a deployment of several access points is typically managed by one or more Wi-Fi controllers, which coordinate configuration, monitoring, and software updates among all of the access points. Like the Wireless-Link Layer, this layer uses the 802.11 protocol as well as traditional wired-network protocols such as Ethernet (effectively acting as a bridge between these two protocols). Typical faults in this layer include resource exhaustion at the AP (due to a high number of simultaneous clients), failure of a wired-network interface, and misconfigurations.

Wired-Backbone Layer. This layer contains the physical cabling that establish connectivity between APs and controllers, as well as the switches and routers that connect multiple wired segments within the network. Typical protocols used in this layer are Ethernet. Common faults here include the failure of the wired physical medium, such as a damaged Ethernet cable, hardware failure in a switch or a router, or the misconfiguration of a switch or a router.

Network-Services Layer. This layer comprises core network services such as DNS, DHCP, web proxying/caching, and captive portals, and their corresponding application-layer protocols. These services must be working correctly in order for an application running on top of the Wi-Fi network to function. Common faults here include failures of the hardware running the service, misconfigurations, and overload due to a high number of concurrent connections to the network.

Content-Services Layer. This layer encompasses all of the hardware and software that serves the application-layer content that will be consumed by users, such as text, images, and video. These services must be

working correctly in order for users to obtain content, and that content must traverse all of the lower layers of the architecture to arrive at the mobile device. This layer contains the services that provide content to the mobile application, and therefore uses the same protocols as the Mobile-Application Layer (e.g. HLS). Common faults here include failures of the hardware running the content services, misconfigurations, overload due to high demand for content, and errors in the content itself (e.g. video-encoding errors).

Because faults may occur at any layer of the system, a robust problem-diagnosis system should diagnose faults at all of these layers. This dissertation focuses on faults from the first three layers (mobile-application, wireless-link, and wireless-infrastructure). Beyond the wireless-infrastructure layer, the infrastructure is closely monitored and often has redundancy, so while faults occur they are rarely user-visible (and, if they were, they would cause a global network outage). Instead, we focus on diagnosing local problems involving access-points, the RF environment, and end-user devices, which are more numerous, more difficult to identify and diagnose, and for which there do not exist robust mechanisms for automated diagnosis and mitigation in most Wi-Fi networks today.

1.3 Thesis Statement

This dissertation explores the following hypothesis:

> *By developing a diagnostic approach that combines performance data from multiple perspectives of the Wi-Fi network, we can quickly detect user-visible performance problems, accurately diagnose faults across multiple layers of the network, and apply automated mitigation strategies for many of these faults in production high-density Wi-Fi networks.*

Specifically, in this dissertation we develop a problem-diagnosis system for large-scale, high-density Wi-Fi networks that:

- Defines a comprehensive fault model across all layers of the Wi-Fi network.
- Instruments end-user devices and access points within the Wi-Fi network in order to obtain performance metrics from those devices.
- Collects and aggregates those performance metrics in real time.
- Analyzes those performance metrics in real time and determines when user-visible performance problems are occurring.
- When a problem is detected, uses data collected from multiple perspectives of the Wi-Fi network to determine the most-likely diagnosis of the problem.

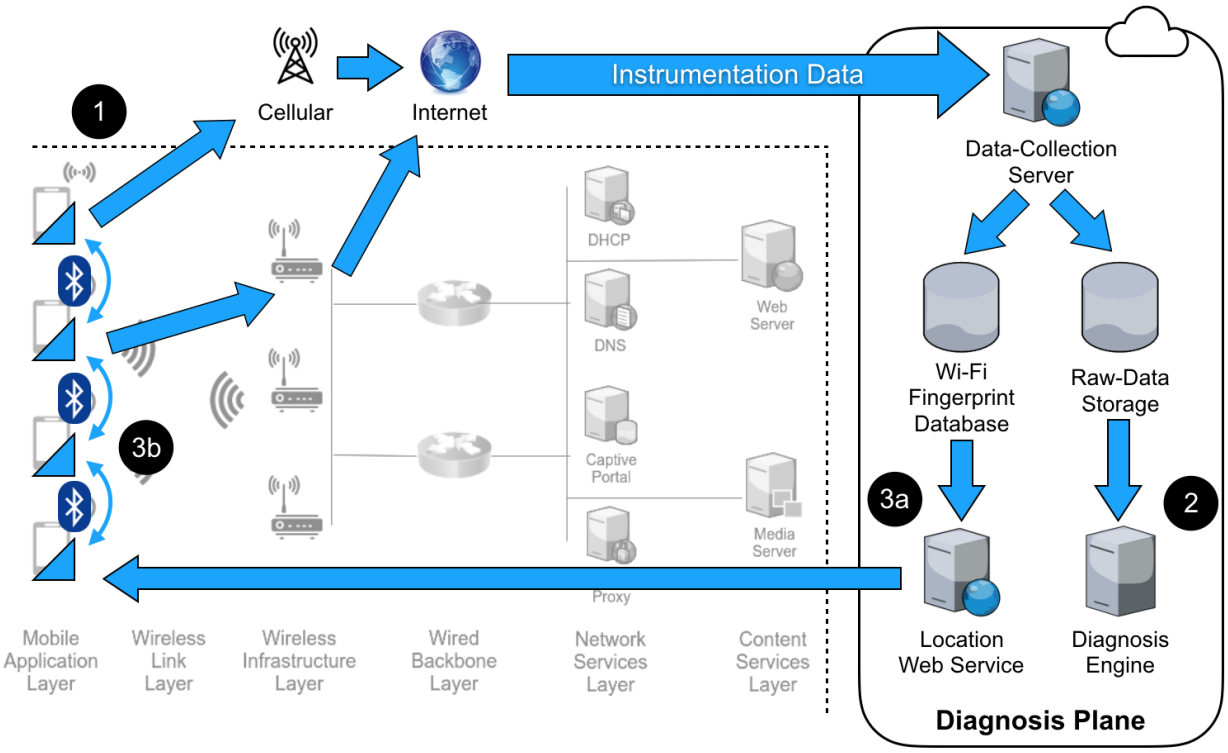

Figure 1.5: A diagram showing the high-level architecture of our instrumentation, data-collection, diagnosis, and mitigation system, and how it relates to the Wi-Fi network under study. The instrumentation on each mobile device is shown as the blue triangles. Data is collected from mobile devices and sent to a cloud-based data-collection system (1) through either a cellular or Wi-Fi connection. Once collected, this data is used to drive both offline diagnosis (2) and location-aware mitigation (3a). We also implemented mitigation strategies using data from multiple perspectives, and a device performing mitigation collects data from other perspectives using Bluetooth (3b).

- Based on the diagnosis of the problem, determines and applies mitigation strategy that will likely resolve the problem for the affected device.

1.3.1 Construction of System

To construct such a system, we first study the behavior of the systems that compose the Wi-Fi network as well as the behavior of the users of the network, as both are important for successfully diagnosing visible problems that users face before they disengage from using the network or the applications that rely on it. As we will describe in more detail later in this dissertation, we start by studying key **instrumentation** points available on the devices in the Wi-Fi network, and how to efficiently and in real time collect data from those instrumentation points. We then combine this data with instrumentation of mobile-application user-interfaces, allowing us to obtain data on user behavior alongside the Wi-Fi performance data and correlate the two in time. We also develop a **problem-detection** approach that allows us to determine

when user-visible problems are occurring, in real time, which in turn allows us to investigate measured Wi-Fi performance at the time when these problems occur. Our study of this data leads us to our first key result in how Wi-Fi performance data affects user behavior, and allows us to set a goal for the latency of our diagnosis.

Next, we use the performance data collected from Wi-Fi devices to perform **problem diagnosis**, including problem detection, localization, and root-cause analysis. Extending our instrumentation and data-collection framework, we introduce a measure of device location into our data, along with the device's proximity to other devices on the network through the use of Bluetooth communication and Wi-Fi access-point scans. We determine a device's proximity with other devices using two methods: Wi-Fi fingerprinting [28] and the exchange of data via Bluetooth [33]. Using our Wi-Fi performance metrics combined with location, we then use both rule-based and decision-tree techniques on the performance data to effectively discriminate between different faults (at the mobile-application layer, the wireless-link layer, and the wireless-infrastructure layer) that manifest as performance problems. Thus, this portion of the system provides us with a most-likely diagnosis of problems when they are detected in the Wi-Fi network.

Finally, we use the diagnosed problem as a basis for selecting a **mitigation** strategy, along with the performance data from nearby access points. For most faults, mitigation is the selection of a different access point that will provide an alternative connection, although in some cases the only possible mitigation option is to physically move the device to a different location (which cannot be automated). Alternative-access-point selection is done by ranking nearby access points based on their proximity to the device as well as the observed performance between each access point and the devices connected to it. If possible, the device applies this mitigation strategy and then continues monitoring for user-visible errors, thus repeating the cycle.

Offline versus online components. In this thesis, some of the components are executed online (while the system is in operation), while others are executed offline. We perform our instrumentation and data collection in an online manner, as we need to collect data about problems while those measurements are available. We also perform mitigation online, because in order to determine whether mitigation is successful, we need to attempt mitigation while a fault is occurring. However, our problem-diagnosis approach is offline, and we use the data collected to attempt to diagnose the problem that occurred. Because we do not diagnose root cause online, when we validate our mitigation strategy, we assume that the fault that occurred is a fault that can be resolved through automated mitigation.

Use of Bluetooth. While we use Bluetooth in the implementation of our approach, the focus of this thesis is strictly on Wi-Fi performance problems. We do not include any Bluetooth problems in our

fault model. The motivation for using Bluetooth in our implementation is to enable a device to collect performance data from multiple perspectives of the Wi-Fi network (from other devices) without using the Wi-Fi network itself (which may be experiencing a performance problem).

Other technologies could be used for sharing Wi-Fi performance data without using the Wi-Fi network, such as communication over a cellular network. We chose Bluetooth for the following reasons:

- Bluetooth is readily available on all major smartphone platforms, including Android.
- Bluetooth is a short-range communication technology, which ensures that devices that are able to communicate using Bluetooth are near each other.
- Bluetooth communication is independent of IP-based communication on the device. Specifically, it can be used at the same time as Wi-Fi or cellular networks, without disrupting communication on those networks.

1.3.2 Goals, Non-Goals, and Assumptions

The primary goals of this dissertation are:

- Study Wi-Fi performance problems that occur in production large-scale, high-density Wi-Fi networks and how users react to those performance problems to determine how to define a user-visible problem.
- Detect user-visible Wi-Fi performance problems in a live-video-streaming application.
- Define a fault model that includes common faults across all layers of the Wi-Fi network.
- Localize and diagnose the root cause of Wi-Fi network performance problems within our fault model.
- Automatically mitigate certain faults that can be resolved through software execution alone.
- Implement our approach such a way that it can be applied to production Wi-Fi networks at minimal cost. Specifically, this means that it does not require invasive modification of either end-user or Wi-Fi infrastructure devices (e.g. firmware updates, custom kernels, or modifications to operating-system software) and also does not introduce additional devices into the Wi-Fi network (e.g. air monitors or packet sniffers).
- Validate our approach using data from our fault-injection testbed as well as real-world Wi-Fi environments.

Non-Goals. This dissertation does not aim to:

- Provide diagnostic coverage of all faults. The data available to us may be insufficient to do so.
- Determine the root cause of every problem. Again, the data available to us may be insufficient to do so.
- Provide on-line or real-time diagnosis of problems. Some problems may require training of models.
- Mitigate every problem that occurs. We may not have access to modify some parts of the system in order to mitigate all problems.
- Provide graceful degradation of service in the face of problems. We may not be able to modify application behavior in order to gracefully degrade service.

Assumptions. This dissertation makes the following assumptions:

- All Wi-Fi faults cause user-visible errors. We don't seek to detect faults that go unnoticed by users.
- No malicious attacks are present. It is easy for a malicious client to impact performance of the 802.11 MAC protocol. We only focus on faults that occur during the normal or expected operation of this protocol.
- The system is free of software bugs. We assume that all software is operating as intended and do not include bugs in our fault model.
- The wired network is not a bottleneck. This allows us to attribute all throughput issues to the Wi-Fi network.
- The Wi-Fi network is infrastructure-based. We do not focus on ad-hoc or mesh Wi-Fi networks.
- End-user devices are Android smartphones running version 7 or higher. The Android operating system provides access to key Wi-Fi performance data that is unavailable from other major smartphone operating systems such as iOS due to restrictions imposed by the operating system vendor.
- Bluetooth and Wi-Fi functionality are enabled on these devices. We require Bluetooth to collect data from multiple perspectives of the Wi-Fi network (nearby peers), and Wi-Fi must be enabled to collect Wi-Fi performance measurements.
- The devices are in close proximity to multiple peers (the device is not isolated). We need data from multiple perspectives of the Wi-Fi network to perform problem diagnosis, and our approach obtains

Wi-Fi performance data from these perspectives through short-range Bluetooth communication with nearby peers.

- All clocks across all devices are synchronized to the same time source. We combine data from multiple network peers by correlating that data in time, using timestamps recorded locally on each device.

- The end users are using a live-video-streaming application, and any errors in the application are caused by Wi-Fi performance problems. We instrument a live video-streaming application in order to detect Wi-Fi performance problems.

1.4 Dissertation Structure

The remainder of this thesis is divided into nine chapters, seven of which (chapters 3 through 9) describe in detail the construction of various components of our problem-diagnosis system. Chapter two discusses related work, while chapter ten summarizes our results into a conclusion and discusses future work.

Chapter 2 discusses related work in the diagnosis of performance problems in Wi-Fi networks. Not all of the related work focuses specifically on large-scale, high-density environments, and not all of the work includes problem-detection, problem-diagnosis, and mitigation components. While this dissertation covers all three of these topics of study, we draw upon and compare our work to other work that may only focus on one or two of these topics at a time. Thus, we divide our related work into sections based on these topics, and discuss the related work in that specific area and how our work compares to contemporary work in that topic of study.

Chapter 3 describes our approach to instrumentation and data collection. This includes our study of the hardware components of a Wi-Fi network and the key instrumentation points that we found in those components. We then discuss client-side instrumentation, which describes our method for obtaining Wi-Fi performance data from client's perspective (the end-user device), along with our instrumentation of the user interface on that device. Next, we discuss our architecture for collecting this data from the thousands of devices that are simultaneously active in real-world large-scale Wi-Fi networks, and how we store and aggregate this information. We then describe our instrumentation of Wi-Fi infrastructure and how we can obtain data from it, and what data is available in commonly-used Wi-Fi infrastructure equipment. Finally, we discuss our strategy for deploying our instrumentation in a non-invasive manner on end-user devices as well as for collecting data from Wi-Fi infrastructure.

Chapter 4 presents our approach to problem detection and how those problems manifest as user-visible video-streaming problems. We begin by describing the operation of widely-used mobile-video streaming protocols, which we use or simulate in various portions of this thesis to create synthetic workloads to study our problem detection and diagnosis. This leads to a discussion of how the video player operates and how we can detect Wi-Fi performance problems by studying the states of the video player along with length of time that the video player remains these states. We then discuss how anomalies in the state machine manifest as user-visible video-streaming problems. Finally, we discuss how we log both the Wi-Fi performance data and video-player state information, and then analyze these logs in order to perform problem detection in real time.

Chapter 5 explores how the user-visible problems discussed in chapter four have an impact on user behavior. Specifically, we study the correlation between user-visible errors and anomalies with user engagement with our candidate video-streaming application. Through this study, we identify key thresholds for how long problem detection can take before a user disengages with the application entirely. This sets our goals for the performance of our system implementation.

Chapter 6 describes our approach to problem diagnosis. This begins with our study of the metrics and attributes that correlate with Wi-Fi performance. We then discuss which of these factors help us discriminate among the possible locations of faults in the Wi-Fi network (at the end-user device, in the RF environment, or at the access point). We then introduce our fault model and the six faults we seek to diagnose as root causes. We then move on to discussing key indicators of faults, and the metrics and patterns in those metrics that are indicative of each of the faults. We then describe our approach to combining multiple perspectives of the network, including how we obtain data from these perspectives and how we aggregate this data to perform problem diagnosis. Once this data is gathered, we finally discuss our diagnostic approach based on pattern-matching rules that we have developed from our study of the data, and how we can set the thresholds and other parameters for these rules by analyzing the data collected from real-world Wi-Fi networks.

Chapter 7 describes our approach to problem mitigation, once a diagnosis of the problem is determined. We begin by discussing the default problem-mitigation strategy present in every Wi-Fi device, which is controlled by the Wi-Fi driver and is usually (but not always) applied when signal strength to the current access point becomes poor. We then discuss how we can override the driver's roaming decisions, including the limitations of what is possible today on Android devices and how this could be improved through operating-system changes. We then discuss our approach to searching for alternate access points that may provide better performance for a device experiencing problems, including our work with fine-grained location tracking through Wi-Fi fingerprinting. Finally, we describe our approach to automatically

selecting an alternate access point based on predicted performance in order to mitigate the problem.

Chapter 8 presents our experimental evaluation of our system. We begin by describing our testbed, our synthetic workload, and our fault-injection strategy. We then explore our the latency of our problem-detection approach, which is critical to keeping users engaged with our candidate application. Next, we study the performance of our diagnostic approach under controlled (laboratory) conditions, to understand the performance of our system in discriminating between injected faults. We then study how the number of devices in the network (peers) affects the accuracy of our diagnosis results. Next, we move our system into a real-world dynamic Wi-Fi environment, and explore the performance of our diagnosis approach using data collected from this environment. Finally, we discuss the effectiveness of our mitigation strategy in terms of how often our strategy successfully mitigated the problem experienced by the user.

Finally, chapter 9 presents our conclusion and future work. We summarize the results of our studies and review the extent to which we have met the goals we outlined in our thesis statement. We then describe possible opportunities for future work and various branches that could be taken from the work begun in this thesis.

1.5 Contributions

My dissertation makes four main contributions that are described in the following sections.

Client-side and infrastructure-side instrumentation for Wi-Fi-performance measurement of production systems. First, we develop an instrumentation system to collect real-world cross-layer performance data from both the client and infrastructure perspectives. This work provides a system for instrumenting mobile devices and mobile applications (the clients) to collect data that is relevant to wireless-performance measurement and diagnostics. Because this system requires that an application be installed on mobile devices to collect data from the operating system and the application itself, this is a white-box measurement approach. However, the approach is not considered invasive, as it is embedded as a library in an application that can be easily installed on the smartphone. This library could also be embedded in any other application in order to instrument that application. The instrumentation of mobile devices is augmented with data from the wireless infrastructure by interfacing with existing monitoring systems, allowing the system to collect data from both the client and infrastructure perspectives. This instrumentation system has been deployed in a real-world mobile application and we have collected data from real-world high-density wireless networks. We use these real-world traces to determine the impact of wireless-network problems on user behavior and as a basis for the construction of an end-to-end fault model and diagnostic algorithm. To our knowledge, we are the first to obtain real-world client-side instrumentation data from

production large-scale, high-density Wi-Fi networks and use this data to determine when user-visible problems occur.

End-to-end fault model for Wi-Fi-network applications. Second, we develop an end-to-end fault model for Wi-Fi network problems that encompasses all components of the network. This fault model will be the basis for our fault diagnosis, and attempts to cover the set of all possible faults that can impact the performance of an application that uses a Wi-Fi network. The fault model includes device configuration and load faults, RF environment faults, AP configuration and load faults, infrastructure hardware and link faults, network service faults, and application-layer content-service faults. The construction of this fault model draws upon related work in our research area, as well as first-hand results collected from our instrumentation framework deployed in real-world Wi-Fi environments. To our knowledge, we are the first to construct an end-to-end fault model for Wi-Fi networks that includes faults from all layers of the system.

Low-latency problem detection, low-cost diagnosis, and automated mitigation for Wi-Fi networks. Third, using our real-world end-to-end traces, we extend our instrumentation framework to detect when problems are occurring using data from the live-video-streaming application. We aim to do this with near-real-time latency (within seconds), so that we can trigger diagnosis as soon as possible and inform the user that diagnosis is in process. We also develop a low-cost diagnostic approach that can be implemented on any smartphone with Bluetooth support running a mobile application integrated with our library. This approach is low-cost as it does not require any specialized infrastructure and does not require invasive modification of devices in the Wi-Fi network. We also describe a diagnostic algorithm that will identify the most likely causes of these problems using data obtained from Wi-Fi network peers. Once this diagnosis is obtained, our approach attempts automated mitigation of the problem, switching to a nearby suitable access point if the problem can be localized to an AP or wireless channel, or by forcing a configuration change on the device itself if the problem is localized to the device. To our knowledge, we are the first to introduce a system that combines low-latency problem detection, low-cost diagnosis, and an automated mitigation system for problems that occur in large-scale, high-density production Wi-Fi networks.

Empirical validation of approach. Fourth, and finally, evaluate the effectiveness of our problem-detection, problem-diagnosis, and automated-mitigation strategies. We do this using both a laboratory testbed designed to replicate the faults in our fault model, as well as through the injection of faults in a real-world arena Wi-Fi deployment at the UPMC Lemieux Sports Complex in Cranberry, PA. We use a synthetic video-streaming workload to replicate the typical low-latency/high-bandwidth workloads seen in real-world Wi-Fi networks. We evaluate the latency of our problem-detection approach, the accuracy of our problem-diagnosis approach, and the success rate of our automated mitigation approach.

1.6 Limitations

Our diagnostic approach encompasses many common and frequent faults that occur in large-scale, high-density Wi-Fi networks. However, our approach does not aim to cover all possible faults and work with all possible Wi-Fi network configurations and devices in those networks. This section specifically describes the major limitations of our approach, and also presents opportunities for extending the work in this thesis to address some of these limitations.

Our approach is intentionally limited to end-user devices on the Android platform. This platform was chosen because it provides access to many key instrumentation points by software installed as applications, thus not requiring any "rooting" or operating-system modification. This was key to achieving our goal of not requiring invasive modification to end-user devices.

We do not attempt to detect problems occurring while the end-user devices are running applications other than the live-video streaming application that we study. We have only instrumented our live-video streaming application for problem detection, so other applications will not provide the instrumentation data that we need for problem detection. Extending our approach to other applications or determining when Wi-Fi performance problems are occurring without application-layer information is a possible subject for future study.

We do not attempt to diagnose all possible Wi-Fi network faults at all layers of the system. Specifically, we do not consider faults in the back-end infrastructure of the network (e.g. cabling, servers, controllers, network services, etc.) as fault-tolerance for these components has been extensively studied, and faults in these components rarely result in user-visible problems in production systems due to the fault-tolerance present in these components. We also do not consider faults due to malicious activity (e.g. frame injection, MAC spoofing, signal jamming, etc.) as these faults have been studied in other literature and these faults are uncommon in the real-world production systems that both we and others have studied.

Finally, we do not attempt to automatically mitigate all faults that we diagnose. For example, configuration faults can be mitigated by automatically changing the configuration of the device, but we do not study that mitigation strategy. We also do not attempt to mitigate faults that require changing the position of devices within the physical space. Mitigating these classes of faults is a possible subject of future study.

Chapter 2

Related Work

The study of failures in Wi-Fi networks has a long history, and has been studied nearly as long as Wi-Fi networks (and predecessors, such as WaveLAN) have been in existence [24]. Much of the original work in Wi-Fi diagnosis can be traced back to similar work on wired networks, and indeed both wired and wireless LANs share many components (and potential faults) in common. These types of faults have been extensively studied, even prior to the advent of wireless LAN technology, for example the work by Maxion and Olszewski [51] on the detection and diagnosis of injected wired-network faults. In that study, the authors explore several faults actually observed in a real-world network. Many of those faults (for example, network paging) are not specific to the wired (versus wireless) nature of the network and could occur in both types of network. In this dissertation, we don't introduce new methods of diagnosing faults common to both wired and Wi-Fi networks; however, we do draw upon the techniques used to diagnose those faults as inspiration for the techniques used in our own work.

Within the study of the faults unique to Wi-Fi networks, there are multiple classes of faults that can occur. Unlike wired networks, where communications signals are contained to fixed pathways shielded from interference and outside influence, signals in Wi-Fi networks are open to influence and modification from anyone in proximity to the network, both intentionally and unintentionally. This introduces a class of faults ultimately caused by malicious activity, and is closely aligned with contemporary work in computer and network security. Often, these faults manifest as performance problems; for example, an attacker may attempt to prevent devices from communicating with the AP by forging invalid network allocation vector (NAV) information, assigning all airtime slots to the malicious client [61]. Another related class of faults are physical-layer faults caused by the configuration of the RF environment itself. Examples of faults in this class are signal absorption and reflection [69], external non-802.11 interference sources (e.g. microwave ovens) [74], and multi-path fading [69]. Finally, we have the class of faults that are

caused by non-malicious workloads that exceed the limitations of the network, such as coverage gaps [14], exceeding channel capacity [73], and AP overloading [77]. This dissertation focuses these faults related to non-malicious device behavior.

Any problem-diagnosis approach must implement the processes of problem detection, localization and root-cause analysis, and problem mitigation. Generally, a problem-diagnosis system will first identify when a problem is occurring (or is about to occur), which will in turn trigger some diagnostic algorithm to determine which fault or faults have occurred, and then finally the system will apply a mitigation strategy appropriate to the fault that was identified. The components that implement these processes are the fundamental building blocks of any problem-diagnosis system, and can be implemented independently of each other. Indeed, some studies only focus on one or two of these components, instead of all three. Therefore, it is constructive to discuss the work related to each of these components individually, as we have done below.

2.1 Techniques for Wi-Fi Performance Measurement

The techniques used to measure Wi-Fi performance fall in to two categories: *active instrumentation* and *passive instrumentation*. These methods differ primarily in how intrusive they are to the system being measured. In active instrumentation, the components of the Wi-Fi network (such as clients and APs) are modified to include code that gathers and reports Wi-Fi performance metrics from the perspective of that device. In passive instrumentation, an independent system measures Wi-Fi performance metrics without any modification of network components by obtaining a copy of the data being passed through the network. Both active and passive instrumentation are used throughout the literature, with active having a recent resurgence of popularity due to the rise of smartphones and other mobile devices.

Passive instrumentation. Passive instrumentation fundamentally involves a Wi-Fi device that continuously receives packets being transmitted over the Wi-Fi network. Typically these are specialized hardware devices that consist of one or more radios and a storage medium. The radios continuously sense the Wi-Fi medium and record packets transmitted on the network, typically along with timestamps so that the logs can be correlated across multiple such devices. Multiple radios may be employed to sense the traffic on multiple Wi-Fi channels simultaneously, or to transmit the logged data to another node for collection.

One example of such a device is a small single-board computer (SBC) that typically runs a Linux-based operating system with one or more on-board Wi-Fi adapters. In the past, devices manufactured specifically for this purpose were commonplace, but such devices can be created today from widely-available SBC platforms such as Raspberry Pi and custom software. They are typically battery-powered,

as they need to be located in areas that may not be adjacent to a power outlet. However, if nearby power is available, they may be plugged in to a wall outlet or use Power over Ethernet (PoE).

Another common implementation of passive instrumentation is to use an auxiliary radio on devices already present in the space, such as access points. This can be accomplished by taking an off-the-shelf device and installing custom firmware which places one or more of the radios in the device into a monitoring (rather than operational) mode. The device can then function as both an operational device (for handling Wi-Fi traffic) and as an air monitor.

There are many research systems that have used this approach of passive instrumentation as part of their data-collection strategy. Airopeek [3] is a specialized Wi-Fi software library used by many systems to capture Wi-Fi packets [50], along with a Wi-Fi interface. Another technique, often used within APs, is to include a second independent radio within the device that continuously senses network traffic [11, 65]. Finally, passive instrumentation can also take the form of a wired-network-side packet sniffer, such as the one used in [15].

Because passive instrumentation requires additional devices that must be purchased, deployed, and maintained, it has a higher cost to deploy than other approaches. For this reason, we avoid using passive instrumentation in our approach, where we aim for an approach that is easy to deploy in production environments by keeping costs low.

Active instrumentation. Active instrumentation involves adding or modifying hardware or software on the critical path of data flows within the network to add measurement and reporting capabilities. This is in contrast to passive instrumentation, which strives to not disturb the operation of the network but simply observe its operation. While active instrumentation necessarily modifies the operational code within devices that compose the network, its primary advantage is the ability to tap into the operational state of the device and log that state in addition to the actual traffic seen on the network. This can lead to much more powerful instrumentation.

By far the most common type of active instrumentation is the customization of AP software to include instrumentation code. This can be done at the driver level [58, 62, 77, 48, 55], at the network-adapter level through packet capture [59, 48], and even through deep-packet inspection to infer the TCP state of each connection passing through the AP [11]. Shi et. al. [68] use Wi-Fi scan and association data reported by the Android operating system to infer the interference graph for a Wi-Fi network. Finally, Dobrian et. al. [23] use active instrumentation of a video player in order to directly detect video-streaming anomalies that may be indicative of Wi-Fi performance problems.

Active instrumentation is the approach we use for our instrumentation and data-collection system. Active instrumentation allows us to piggyback our software alongside other software already present on

devices in the space (in the case of mobile applications), and to use instrumentation already available in existing devices (in the case of Wi-Fi infrastructure).

Data sources and metrics. Whether using active or passive instrumentation, a wide variety of metrics can be collected from the client-side or infrastructure-side perspectives. The metrics are generally common across both types of instrumentation; in active instrumentation, they may be obtained directly from the hardware or software state itself, as the system may already store or compute these measurements for other purposes. Examples of this include signal strength (through the relative signal-strength indication, RSSI [42]), which is often measured by Wi-Fi drivers in order to perform AP selection, and channel utilization, which is often measured by APs to load balance clients and do automated channel selection. In passive-instrumentation systems, these metrics may need to be obtained from fields within packet flows or even computed by analyzing packet sequences.

Because each device within the network (client or infrastructure) has a different role, these devices typically measure and record different performance related metrics. We further subdivide the metrics between these two data sources. First, we have client-side metrics, which form the client-side perspective and typically include detailed measurements of a single wireless-communication channel (which the device is currently using), along with measurements of network-layer and application-layer performance. Second, we have infrastructure-side metrics, which typically provide aggregate measurements across all communications channels managed by a single AP (e.g., total airtime utilization) and perhaps some details about individual communication channels. The metrics studied in the literature for each of these perspectives is described below.

From the client-side perspective, one can gather Wi-Fi scan data, including the network name (the service-set identifier, or SSID), the MAC address of the access point (the basic-service-set identifier, or BSSID), RF channel, and RSSI for each nearby AP [52, 68, 7, 14], details about the current association, including base rate, RSSI, packet counts, retransmission counts, channel, throughput) [52, 68, 61, 73, 14, 55, 10]; TCP statistics, including congestion window size, number of transmissions, etc. [14, 55], packet traces as seen by the client [48, 2, 1], and application-layer statics such as video-buffer state [52, 23].

From the infrastructure-side perspective, we can obtain a wide range of metrics by instrumenting the APs themselves. At the physical layer, these include antenna gain [73], overall noise power and SNR for each client [73], the transmit power of the AP [73], and the receive power from each client (RSSI) [55]. At the 802.11 link layer, these include the number of devices associated [73], airtime utilization statistics [73, 55, 62], packet and byte counters for both transmit and receive [55], the number of transmit failures and retransmissions [55], and traffic statistics for each client [77, 11, 58].

Our approach to collecting data through low-cost active instrumentation of the devices under study

also limits us to the metrics provided by the operating system. This is a trade-off we have made to achieve our goal of a low-cost approach for production networks. Fortunately, the Android operating system provides a rich set of Wi-Fi-performance data that we can collect. However, through more invasive instrumentation of the device (for example, by instrumenting driver code or exposing more metrics from the operating system), it may be possible to obtain even more instrumentation data.

Critique. Passive instrumentation approaches have the advantage that they do not modify the system under study. However, passive approaches can be costly to implement, as they often require the purchase of additional hardware and require human intervention to maintain. Passive approaches also do not have access to deeper diagnostic data that active approaches may have.

Active instrumentation, on the other hand, has access to a richer set of performance metrics. However, it modifies the system under study, so one must be careful to not introduce any anomalies or side effects into the system when introducing active instrumentation. Also, active approaches often require invasive modification of devices in the network, which may not be feasible in production Wi-Fi networks, especially ones where the network operator does not own the end-user devices.

2.2 Fault Models and Diagnostic Approaches for Wi-Fi Performance Problems

This section outlines related work in the area of diagnosing performance problems that occur during the normal operation of Wi-Fi networks. The construction of any diagnostic system rests upon a fault model, which is the set of all faults that are considered by the algorithm as possible root causes. When diagnosing a problem, diagnostic algorithm can be seen as a function that accepts an arbitrary number of inputs (the instrumentation data) and outputs a set of one or more faults that are the cause of the problem. The fault model defines the range of this function, and it may either output a single root cause or multiple root causes for the fault. Furthermore, the fault model defines the set of faults to be injected when evaluating the performance of the algorithm, to determine how well the algorithm is able to discriminate between the faults.

Once the fault model is defined, an algorithm must be constructed to discriminate between the faults in the model given the performance data presented. There are many techniques that can be applied here, including rule-based approaches, decision-trees, and machine learning. In this section, we summarize the most common techniques found in the literature, and highlight their strengths and weaknesses.

Fault models. In this section, we summarize the fault models that other studies have used when diagnosing Wi-Fi performance problems. Table 2.1 includes a selection of representative faults for each layer of the system. Each of these faults has been studied in at least one of the works we reviewed.

Layer	Representative Faults	Description
Mobile Application	Accidental Association	The client has accidentally associated with the incorrect Wi-Fi network, for example a rogue AP.
	Sticky Client	The client is maintaining an association to a distant AP when a significantly closer AP (in terms of signal power) is available.
	Wrong Credentials	The device is unable to authenticate to the Wi-Fi network due to incorrect credentials.
Wireless Link	Coverage Gap	There are no APs in range to which the device can connect.
	Uplink Congestion	There are too many devices attempting to transmit, causing collisions and slowing performance for all devices.
	Downlink Oversubscription	There is not enough airtime available for all devices to receive data at the bandwidth that they require.
Wireless Infrastructure	AP Overload	Too many devices are attempting to use the same AP, causing the AP to breach hardware or software limitations.
	Firmware Bug	A bug in the AP's firmware limits the performance of devices connected to that AP.
	Excessive Buffering	The configured interval between power-save buffer transmission is too large, causing latency or packet loss.
Wired Backbone	VLAN Misconfiguration	A wired-network port for a Wi-Fi device is configured for the wrong VLAN.
	Firewall Misconfiguration	A firewall is blocking packet flows needed for Wi-Fi traffic.
	Cable Damage	A network cable is damaged, causing complete failure of the wired link or slow performance.
Network Services	DNS Crash	A DNS server is down or slow to respond to queries.
	DHCP Crash	A DHCP server is down or unable to grant IP address leases.
	RADIUS Crash	A RADIUS authentication server is down, preventing clients from authenticating to the network.
Content Services	TCP Backoff	TCP slows network connection incorrectly assuming packet loss due to network congestion rather than wireless reception problems.
	Upstream Bottleneck	WAN connectivity to content services is slow or down.

Table 2.1: Representative faults for each layer of a Wi-Fi network.

Much of the literature has been focused on faults in the RF environment itself. WiSlow [48] addresses broadband interference, spread spectrum interference (non-802.11), channel contention (same AP), and co-channel interference (across multiple APs). MOJO [66] focuses on hidden terminal scenarios, capture effects, general noise and interference, and fluctuating signal strength. Similarly, Singh et. al. [69] identify interference, signal absorption and reflection, multipath fading, and hidden terminal scenarios as possible causes of failures in the wireless medium.

Outside of the RF environment, there are a variety of faults that can affect the ability of Wi-Fi clients to use the network. These range from client-configuration errors to server failures. Many approaches focus primarily on security problems, such as unauthorized networks (e.g. rogue APs), MAC spoofing, MITM attacks, denial of service, packet injection, phishing, and various attacks on Wi-Fi session keys [69, 77, 7]. Others include service failures outside of core Wi-Fi infrastructure that affect with Wi-Fi clients, including overloaded RADIUS and Active Directory servers, misconfigured VLANs, damaged cabling, MTU configuration and discovery issues, upstream bandwidth bottlenecks, DNS resolution problems, broadcast protocols causing packet storms, DHCP failures, WAN congestion, firewall/proxy failures, and hardware failures [11, 14].

The faults most related to the ones we have studied in this dissertation are those in the mobile-application, wireless-link, and wireless-infrastructure layers. Faults in these layers include those related to driver behavior, such as sticky clients (where the driver maintains an association to a distant AP when a closer AP is available) [68] and Wi-Fi driver bugs (where the driver is unable to establish or maintain a connection to the Wi-Fi network) [14]. Wireless-link faults include signal coverage and channel capacity, such as coverage gaps [1, 14], channel contention [48], and channel oversubscription [73].

Diagnostic approaches. Much of the work on problem diagnosis for Wi-Fi networks has used rule-based approaches, either looking for patterns or executing statistical tests on metrics. DAIR [7] uses an inference engine, which executes a set of tests in response to a detected problem and outputs which tests failed as possible causes. Adya et. al. [1] combine the data from multiple peers together and use peer comparison to isolate failures based on rules and thresholds. Shaman [16, 17] uses a sequence of diagnostic tests, where the result of the previous test indicates which one to run next.

Recently, some approaches have leveraged machine learning, primarily decision trees, to diagnose faults. Pei et. al. [59] use decision trees based on transmit physical rate, receiving physical rate, percentage of packets retried, and RSSI to diagnose transmit-power and interference faults. They found that these four metrics had the highest relative information gain of several that they studied. WiFiSeer [73] also uses decision trees based on channel utilization, number of clients connected to an AP, and interference power to select an AP with the lowest predicted latency for each device in the network.

Critique. The literature we reviewed has covered the diagnosis of a wide range of faults that can occur in Wi-Fi networks. The number of possible faults is vast, and it is possible to break down many faults into even more granular root causes. No one system today covers the full gamut of faults that can occur.

Diagnostic techniques fall into two major categories of rule-based or learning-based. Rule based approaches are conceptually simple to understand and construct, and often perform well when tested against the Wi-Fi network for which they were developed. However, they tend to be brittle and difficult to apply to other Wi-Fi networks, as the rules inherently contain parameters and thresholds that have been tuned for a single Wi-Fi network. Learning approaches help to reduce some of this brittleness by providing a way to automatically determine the appropriate thresholds for a given Wi-Fi network, and also to learn which performance factors best discriminate between faults.

2.3 Mitigation of Wi-Fi Performance Problems

Once a problem is diagnosed and the most likely root cause fault is known, the system can take many different actions to attempt to clear the fault or at least work around the problem by taking some mitigating action. Ideally, such mitigation would be automatic. This can be accomplished for some faults, such as channel oversubscription, if another nearby AP is available on a different channel. Similarly, if a Wi-Fi driver is being sticky, the mitigation strategy can be to force the driver to connect to the nearby AP. However, not all faults can be resolved automatically. It may not be possible to mitigate the above faults if another suitable AP is not available. In the latter case, the sticky-client fault becomes a coverage-gap fault if no other AP is available. The only suitable mitigation strategy for this fault is to request that the user physically move the device in range of an access point. If there is a widespread issue affecting a large number of devices, automated mitigation strategies may only move the location of the problem or even make the problem worse (by affecting more users than were originally affected). In this case, the only option is to involve a human operator to manually resolve the problem.

Several diagnosis systems that we studied employ automated problem-mitigation strategies. WiFiSeer [73] uses its mapping from access points to predicted latency to automatically override the client's AP selection to use the AP with lowest latency. Dyson [55] attempts automated mitigation on the infrastructure side in ways that only affect a subset of clients by either determining the best AP for each client (and forcing the client to connect to that AP), reserving airtime for VoIP and other clients with low latency requirements, and balancing uplink and downlink traffic across APs by forcing clients to associate with other APs until the network is balanced.

Manual mitigation strategies are more common to address wide-area performance problems. ReTiMon

[22] exposes a real-time monitoring dashboard, allowing network administrators to view performance measurements. Shaman [16, 17] provides a tool for end-users to report problems and receive the most likely diagnosis of the problem. Shaman also provides automated alerts to administrators of likely problems with network services, such as DHCP, based on packet traces. WiFiProfiler [14] presents a GUI to the user showing the status of the diagnosis process. Akella et. al. [4] provides suggestions for mitigation of the problems that it diagnoses, including optimizing channel allocation or reducing power on APs.

Critique. Automated diagnosis systems work, but one must be careful to not introduce further problems or cascading failures. For example, if an AP becomes overloaded or the channel it is on is at capacity, an automated mitigation strategy may force most or all of the devices on that AP to instead associate with other nearby APs. This, in turn, may cause the same problem to occur on those APs, causing further re-associations, and so on. Limiting the rate at which automated mitigation can be applied may reduce the risks of some of these side effects.

Chapter 3

Instrumentation and Data Collection

In order to understand the nature of problems in Wi-Fi networks, first it is necessary to collect data about them. The first step in developing our problem-diagnosis system is to build an instrumentation and data-collection system that will serve as a solid foundation for the remainder of our approach. To gain insight into the types of data available, we studied the instrumentation point available on major smartphone platforms as well as infrastructure equipment from a major Wi-Fi vendor.

This chapter describes our approach to and the construction of our instrumentation and data collection system. Specifically, we cover the key instrumentation points of a Wi-Fi network, our white-box instrumentation of the Android mobile operating system to obtain Wi-Fi performance data, our data-collection strategy that allows our approach to scale our data collection across thousands of real-world devices, our approach to instrumenting and collecting data from Wi-Fi infrastructure, and finally close with a discussion of how our instrumentation library is deployed in a non-invasive manner, allowing us to collect real-world data from production deployments.

The goal of our instrumentation and data-collection system is twofold. First, we aim to collect data from actual users of large-scale, high-density Wi-Fi networks in order to characterize user behavior in the face of Wi-Fi problems, and also to identify the metrics that correlate with those problems. Second, we aim to provide this data as an input to our problem-diagnosis algorithm, which we will use to ultimately diagnose the root causes Wi-Fi performance problems.

3.1 Wi-Fi Network Devices

We start with a description of the devices that compose a Wi-Fi network, upon which we build our instrumentation strategy. These devices fall into two main classes: end-user devices and Wi-Fi infrastructure. Typically, Wi-Fi infrastructure devices are placed in fixed locations throughout the physical environment.

They are often connected to a wired network for remote access and to serve as a bridge between the end-user devices and wired devices, such as servers. For a given Wi-Fi network, all of the Wi-Fi infrastructure devices are made by the same manufacturer. End-user devices are typically more varied in terms of make and model, as users often purchase these devices independently, bring them to the location of the Wi-Fi network, and connect them to the network. These devices are also much more mobile than Wi-Fi infrastructure, as laptops can be used anywhere there is a flat surface, or even the user's lap, while mobile devices are typically carried in the users' pockets as they move throughout the space. These two classes of devices communicate using the 802.11 protocol [26], which involves the transmission of wireless signals using multiple radios contained in each device. We now describe the operation of these devices in more detail.

End-user devices. End-user devices come in two main flavors: mobile devices and laptops. As mentioned above, mobile devices are typically more mobile than laptops, and also are used in places where laptops typically are not used, such as in sports and entertainment venues. Although they are smaller and more portable than laptops, mobile devices have very similar capabilities to laptops in terms of the nature of their network communication and workloads. Users often stream high-definition video to their smartphones while connected to Wi-Fi networks, and modern smartphones are more than capable of playing back high-definition video, with some even capable of 4K video playback.

Because mobile devices can be used in more locations than laptops, are equally as capable of generating low-latency/high-bandwidth workloads as laptops, and because our candidate application is a video-streaming mobile app, we chose mobile devices as the class of end-user device to study in this dissertation. That is not to say that our approach could not be extended to apply to laptops as well, but this is reserved as an exercise for future work.

The two dominant operating systems in use on smartphones today are Google's Android operating system and Apple's iOS operating system, and recent reports show that 99.6% of new smartphones manufactured today run one of these two operating systems [75]. Currently, the market share for the two platforms are approximately tied in North America (52% iOS to 48% Android) [70], but Android holds a dominant advantage over iOS worldwide in terms of market share (72% Android to 23% iOS) [71]. The iOS operating system is designed to function specifically with Apple-manufactured hardware devices, specifically the iPod, iPhone, and iPad hardware lines. The Android operating system, on the other hand, is available on a much wider variety of devices from several manufactures, all of which vary significantly in hardware specification.

The primary difference between iOS and Android, for the purposes of Wi-Fi instrumentation, is the availability of instrumentation points for Wi-Fi performance measurement in a non-intrusive manner (that

is, without modifying the hardware, firmware, kernel/drivers, or operating-system software). We studied the data available on both platforms. Android is by far the platform most amenable to instrumentation, as it provides a set of classes in the standard Android runtime library specifically designed to provide detailed information about Wi-Fi status, measurements, and statistics [34]. In addition to programmatic access to Wi-Fi information, Android also provides common system metrics such as memory and CPU usage through the /proc filesystem common on Linux-based devices [64], and also provides a method for instrumenting the stock video player to collect application level metrics around video playback [38]. iOS, on the other hand, is much more restrictive in terms of the data available through standard operating-system libraries. It is possible to obtain basic Wi-Fi connection information, such as the currently-connected SSID and BSSID, but other Wi-Fi information is unavailable to user-mode applications. It is possible to obtain application-level performance metrics by instrumenting the video player [5]. Beyond this, access to performance data requires jailbreaking the device in order to access private APIs . If this were not a restriction, it would likely be possible to access a large amount of the same information as is found on Android, perhaps even more detailed information, as many different Wi-Fi-related state variables and counters can be found in the unofficial iOS private framework documentation [46]. Because of the data readily available on the Android platform, it's prevalence worldwide, and its wide support for different device models, we chose to instrument the Android platform in this dissertation.

Wi-Fi infrastructure devices. Wi-Fi infrastructure devices, for the purposes of this dissertation, are the access points that the end-user devices associate with in order to establish a Wi-Fi connection, bridging the device to a wired network that typically provides network and application services and likely an Internet connection as well. Like smartphones, there are multiple manufacturers of Wi-Fi infrastructure equipment, including Cisco, HP, Aruba, Ubiquiti, Extreme Networks, Riverbed Xirrus, Ruckus, Samsung, Motorola, and others. Over the past five years, Cisco has been by far the most dominant manufacturer of this equipment in the market place, consistently capturing around 50% of the market share [72]. Many of the Wi-Fi platforms provide some non-intrusive access to Wi-Fi performance metrics, through some form of an API. We researched the Cisco Wi-Fi platform and found numerous metrics available through the Cisco CMX Controller [21] and the Cisco Prime Infrastructure platform [20]. Also, based on our study of production Wi-Fi networks, we estimate that about 60% of Wi-Fi networks in stadiums today use Cisco infrastructure. Because of Cisco's dominant market share and the availability of APIs providing infrastructure-side instrumentation points, we chose to instrument the Cisco enterprise Wi-Fi platform.

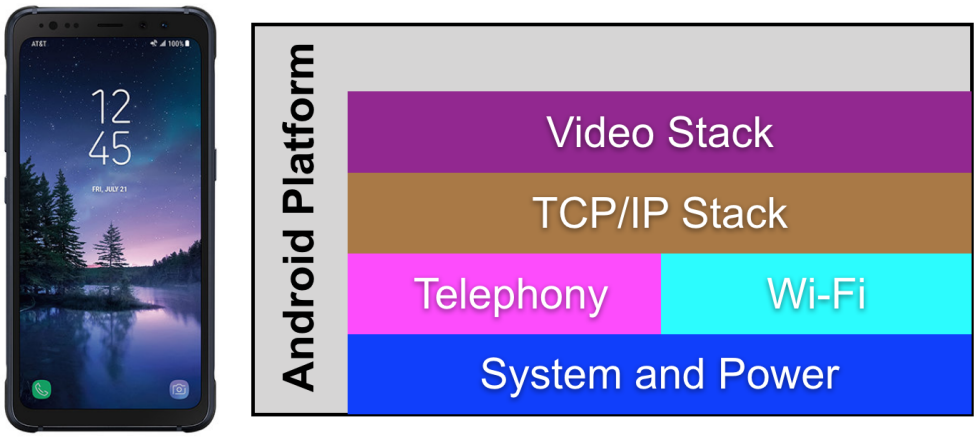

Figure 3.1: A diagram showing the layers (subsystems) of the Android platform that are used during video playback. We instrument each one of these layers to collect data for our system.

3.2 Android Instrumentation

In this section, we aim to describe in detail the key instrumentation points that we found through our study of the Android platform and the metrics that it can provide. Through the platform API, we are able to obtain data on the performance of the Wi-Fi network, the components that manage the Wi-Fi connection, and other subsystems that assist in the operation of the device, specifically for video playback. A diagram of these subsystems and how they relate to each other is shown in Figure 3.1. Note that a discussion of the reliable collection of data from multiple perspectives is reserved for the following section.

On each instrumented device, the data obtained through our instrumentation is recorded in a log with each log entry containing a timestamp and an arbitrary set of data. Each entry in this log represents some change in the state of the device, a change in the Wi-Fi environment, or an updated measurement. This log allows us to concisely record the sequence of changes that occur in the client-side perspective over time, avoiding unnecessary duplication of data. Note that we can reconstruct the entire state of a given device at any point in time by scanning the log entries for that device in time sequence from the beginning and setting various state variables as the log entries are read.

Wi-Fi scanning. Android periodically triggers scans of nearby Wi-Fi networks in order to find nearby access points with which to associate and to display the list of available networks to the user. This process works by periodically pausing normal Wi-Fi operation and hopping between the Wi-Fi channels within the band while listening for 802.11 beacon messages from APs. To minimize disruption to normal traffic flows, Wi-Fi drivers often schedule scans while in powersave mode, meaning that the AP will not be sending any data to the client during that period (powersave mode is so named because the client has the option of shutting off power to its radio during that period to conserve energy).

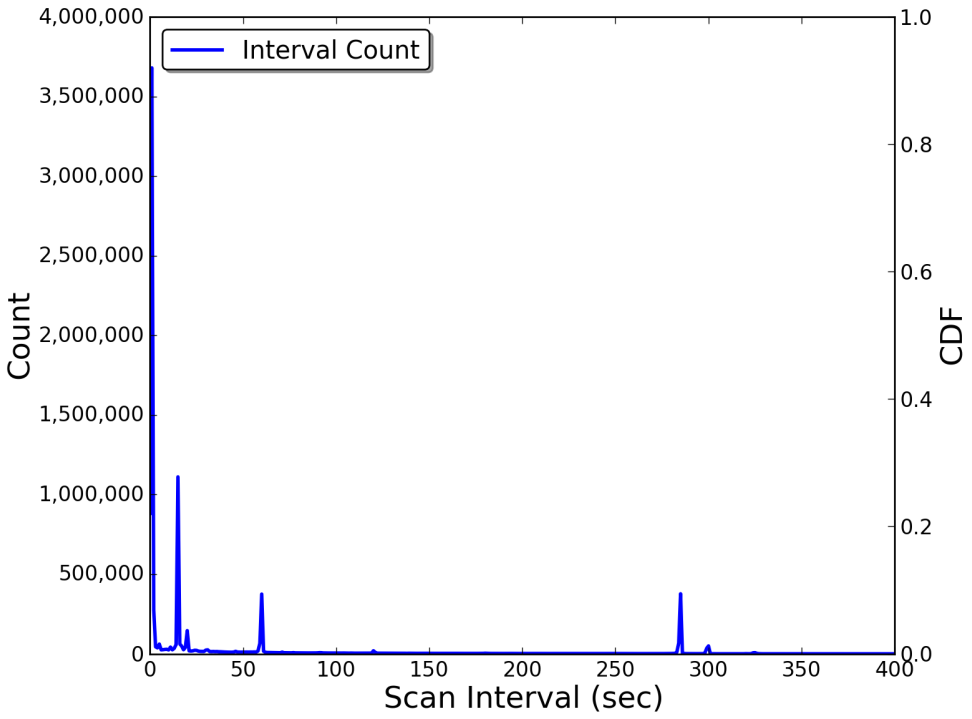

Figure 3.2: Scan interval distribution for Samsung-manufactured Android smartphones over a period of 5 months [67]. The y-axis shows the number of scans recorded as delay since the last scan increases (along the x-axis) for 100 devices over a 5-month period. Android devices scan frequently while the device is disconnected (peaks at 0-1 and 15 seconds), but this slows to 4.5 minutes once the device is connected.

We can make use of the device's Wi-Fi scans to understand the client's perspective of the network. Instrumenting these Wi-Fi scans can be done with the API that Android provides to mobile applications and services. An Android application that has been granted Wi-Fi management permissions can access the latest scan results at any time, but there is no guarantee that those scan results will be fresh. Figure 3.2 shows the distribution of Android scan intervals taken from over 100 Samsung smartphones over a 5-month period [67]. The peak near the origin is the case where the Wi-Fi adapter is enabled but not associated with the AP, in which case the operating system instructs the driver to scan at frequent (typically 15-second) intervals in order to discover APs with which to associate. However, once Android has connected, the scan interval slows down to 4.5 to 5 minutes. This delay between updated scan results is far too infrequent, as our diagnosis algorithm may be working with minutes-old data and not with current results.

To consistently obtain more up-to-date results that will provide fresh data for our diagnosis algorithm, our instrumentation package periodically requests that the Wi-Fi driver conduct a scan of nearby access points at 5-second intervals. We trigger these scans by using the `startScan` API call, which requests that

a scan be conducted at the next available opportunity. Note that this does not guarantee that the device will trigger a scan immediately, but our experience suggests that the scan typically occurs within 5-10 seconds of the request. Just prior to the time of this writing, Google has deprecated this API call in the latest revision of the API; in future releases, this capability will be removed, and some other mechanism, such as WifiScanner [39], will need to be used in order to request high-frequency Wi-Fi scans.

We register a callback with the operating system to inform our data-collection system when new Wi-Fi scan results are available. Once we receive this callback, we call `getScanResults` to retrieve the latest set of results. The set of results is a list with one entry for each BSSID seen, including the RSSI measurement to that SSID and the timestamp of when the entry was last updated. We filter out changes to the scan results by tracking the previous and current set of scan results and, for each item in the current set, if the BSSID was not in the previous set or if the timestamp has updated, we add this as a change to our log. We also include deletions if a BSSID is in the previous list but not in the current list.

The Wi-Fi state machine. The Android operating system and Wi-Fi driver must go through a sequence of several steps in order to successfully establish and maintain a connection with the Wi-Fi network. As it does this, the Android platform exposes information about which portion of this process it is currently executing in the form of a state indicator. The full set of states is shown in Figure 3.3. The black boxes indicate the set of all possible states, with lines in this tree indicating sub-states of each individual state. The red lines in this diagram indicates the states typically traversed while the Wi-Fi stack is initializing and while connecting and maintaining a connection to the Wi-Fi network.

The sequence of states shown here also provides insight into everything that can go wrong (and thus indicate a fault) while maintaining a connection to a Wi-Fi network. The Wi-Fi driver can fail to initialize, in which case the driver attempts to unload and then moves into the "Driver Failed" state. If the driver is successfully started, the driver will move into "Connect Mode" where it will attempt to find an AP to connect with. If no AP for a known Wi-Fi network is available, it will remain in the "Disconnected" mode while doing periodic Wi-Fi scans. If the driver successfully connects with an AP, it enters "L2 Connected Mode". At this point, the Android operating system takes over with higher-level logic that obtains an IP address (via DHCP), verifies the link and checks for captive portals (by attempting to access the Internet), and finally transitions to the "Connected" state at which point the device will begin passing application-layer traffic over the Wi-Fi interface. If any part of this connection process fails (the device is unable to obtain an IP address, the device detects a captive portal, or is otherwise unable to contact a site on the Internet, the device will not use the Wi-Fi interface and will route traffic over cellular instead (if available). Note that, even if the source of the application's data is local to the Wi-Fi network, the device will not use the network unless Internet connectivity is available.

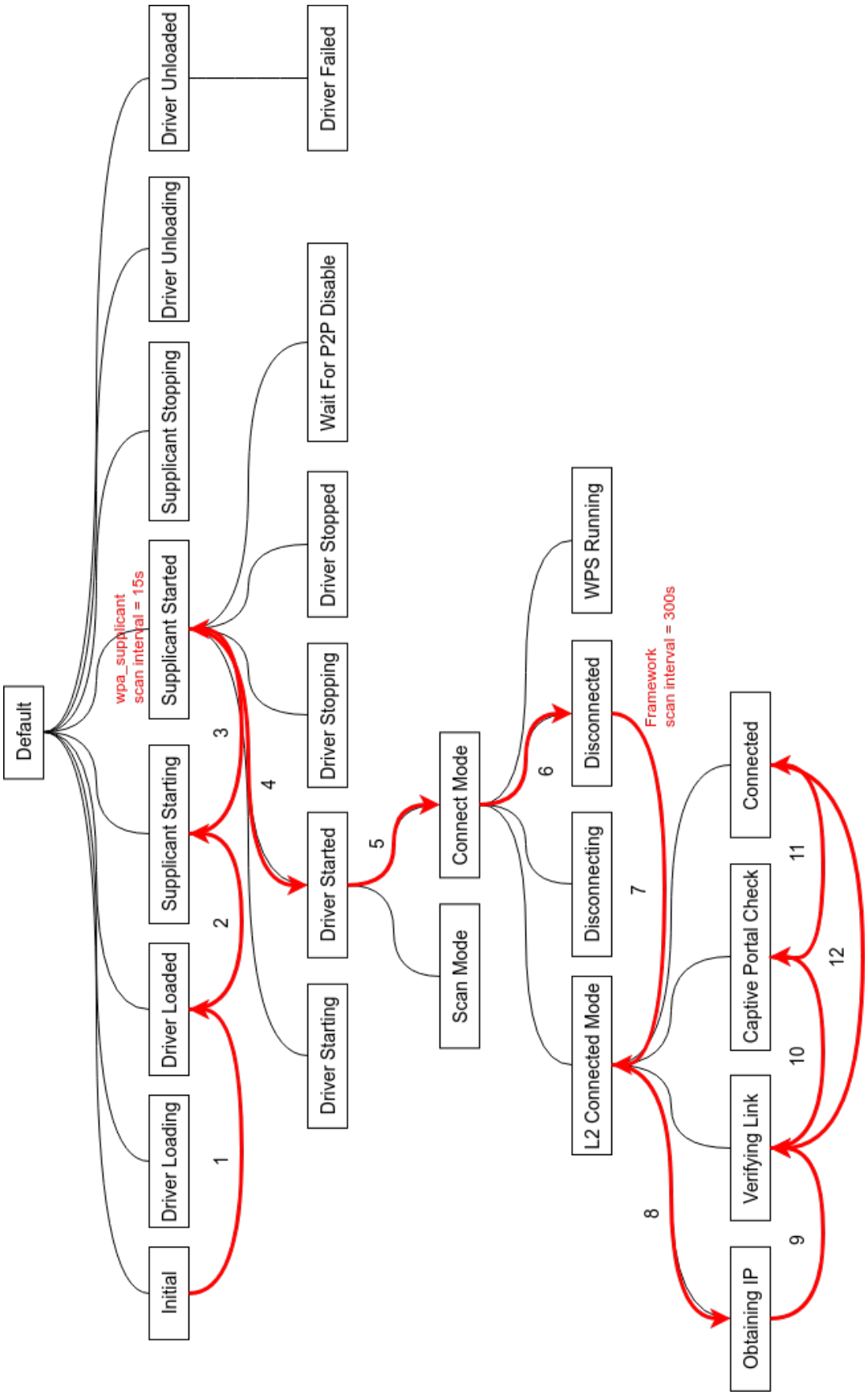

Figure 3.3: A diagram of Android Wi-Fi state machine [67], showing the states that the Android Wi-Fi subsystem can enter. The bold arrows show the sequence of states the driver must transition through in order to successfully connect to a Wi-Fi network. A fault in any of these states may prevent the device from successfully establishing a connection to the network.

Battery/Power Metrics	**System Metrics**
Screen on/off	CPU usage*
Battery charging/draining	Memory available*
Battery charge level	Memory low (yes/no)*
Telephony Metrics	**Wi-Fi Metrics**
Network type	Connection state
Network ID	Wi-Fi enabled/disabled*
System ID	Driver state and failure codes*
Base Station ID	802.11 supplicant state*
Cell ID	Current BSSID
Location area code	Is hidden network
Primary scrambling code	IP address
Data activity state	Link speed (negotiated)
Data link state (up, down)	MAC address
Data network type	Signal strength of current AP (dBm)
Network operator name	Current SSID
Network operator number	Nearby BSSID (AP) count and list
Roaming status (yes/no)	Nearby SSID count and list
Voice network type	Nearby APs in range
Voice link status	Nearby frequencies used
Data signal strength dBm	Nearby AP signal strength stats
Data signal strength ASU	
Neighboring cell count	**Video Playback Metrics**
Neighboring cell signal	Player state
	Is playing (yes/no)
TCP/IP Metrics	Buffering percentage
Bytes received	Playback position
Bytes sent	Playback URI
Packets received	Error description and codes
Packets sent	

Table 3.1: A listing of the metrics collected via our instrumentation of the Android platform. All of these metrics are collected from the client's perspective. Metrics suffixed with an asterisk are metrics that have not been featured in prior work that we have studied.

We instrument the Wi-Fi state machine by requesting that the Android operating system notify our system of any state changes when they occur using a callback mechanism. In addition, our system also polls the current state of the Wi-Fi driver at 5-second intervals just in case any callbacks are missed. To detect changes, we record both the previous and current state, and compare the two any time we receive a callback or poll the current state. If the state has changed, we record this in our log as a change. Using this mechanism, we can reconstruct both the current state of the device at any point in time, as well as the sequence of states that led to that state. As we will see, this will be useful for both problem detection and diagnosis.

Other Wi-Fi metrics. In addition to scan results and Wi-Fi states, we collect a number of additional Wi-

Fi related metrics. Table 3.1 includes the full list of metrics that we have instrumented on the Android platform. Notable among these are the Wi-Fi enabled/disabled state, which indicates whether the Wi-Fi adapter is enabled or disabled. If it is disabled, this is an obvious root cause of any Wi-Fi related issues with the device. Along with the current driver state, if the state is connected, we track the current SSID, BSSID, IP address, MAC address of the interface, and the negotiated link speed. All of these metrics are polled along with the current Wi-Fi state at 5-second intervals.

Telephony metrics. Although not directly related to Wi-Fi performance measurement, our instrumentation also includes metrics around the telephony subsystem, which manages data connectivity to the cellular network and also places and receives telephone calls. Like the Wi-Fi subsystem, Android provides a number of performance-related and non-performance-related statistics around the cellular network. We chose to instrument the telephony subsystem as well, for two main reasons: (i) use of the telephony subsystem (for example, active cellular data links) while Wi-Fi is connected may indicate a problem with the Wi-Fi network, as the device has chosen to route data over the cellular network instead of Wi-Fi. Second, we may want to someday extend our approach to the diagnosis of cellular-network faults, in which case this data may prove useful. The full list of telephony metrics is listed in 3.1. Since many of these metrics are not directly applicable to Wi-Fi diagnosis, we will not describe all of these metrics in detail here, but the interested reader may refer to the Android framework documentation for more details [37].

System metrics. We collect a few specific system metrics in order to allow our diagnostic approach to determine whether there is a correlation between Wi-Fi performance problems and hardware factors. We selected hardware-related metrics in three major categories: power consumption, CPU usage, and memory usage. To measure power consumption, we measure the current battery charge level at 5-second intervals and also record the operating system's indication of whether the battery is charging or draining. To determine power consumption over time, we can look at the rate of change of battery charge level. If the screen is turned on, this indicates a higher rate of power draw, as the screen (specifically, the backlight) is typically the single largest consumer of power in smartphones [13]. To measure CPU usage, we poll the `/proc/stat` file at 5-second intervals, which records the amount of time spent utilizing the CPU in various modes [44]. We compare the previous to the current reading of this file to determine the number of seconds that the operating system spent utilizing the CPU and then divide by the total number of seconds between readings to determine CPU utilization. Finally, we record memory utilization by polling the standard Android API calls `Runtime.totalMemory` and `Runtime.freeMemory`. We also record a flag set by the Android operating system if the Java runtime is in "low memory" state, which triggers additional behaviors (like disposing of background activities) in order to free up memory.

Video playback metrics and state machine. Finally, to determine the user experience at the application

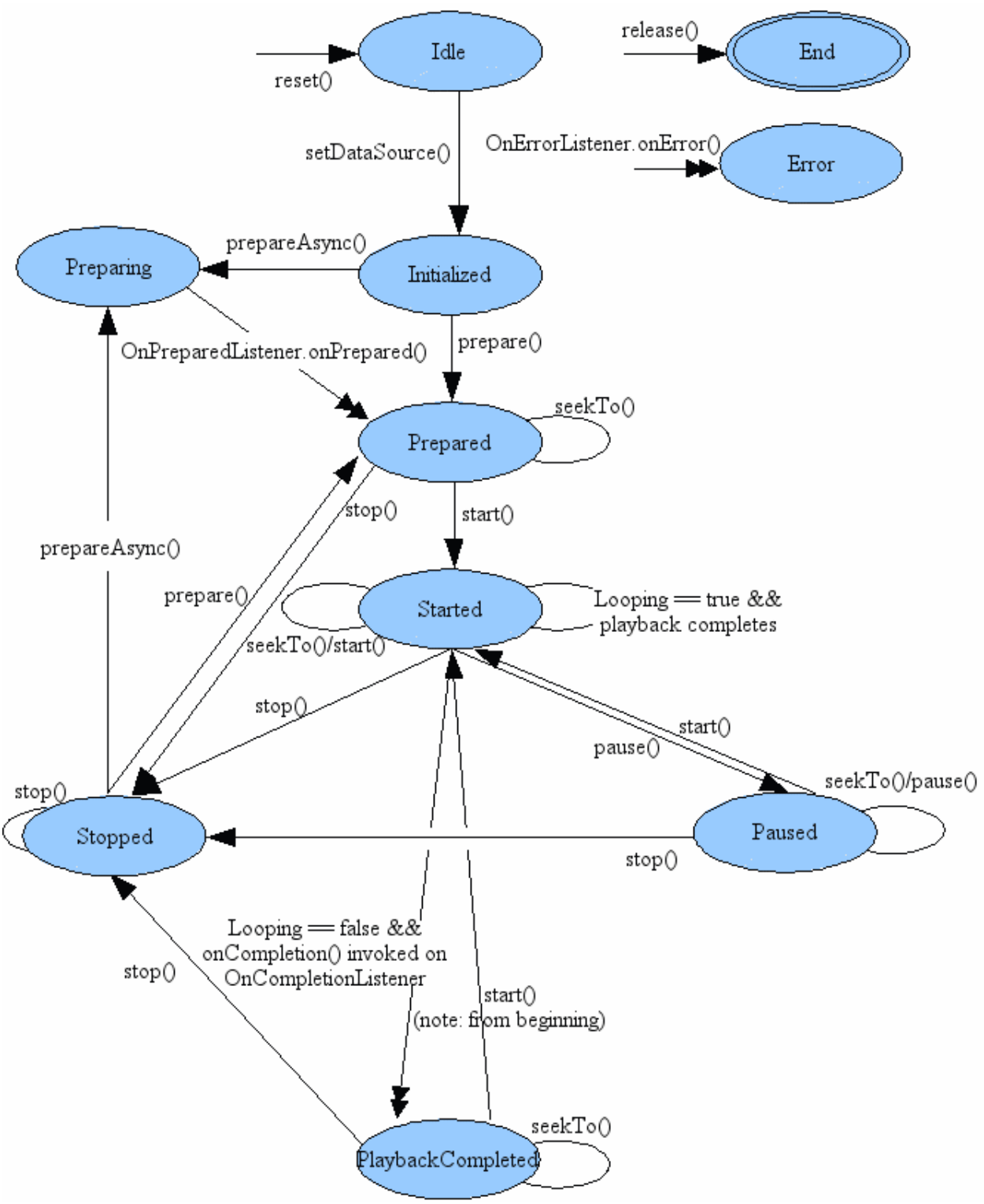

Figure 3.4: The Android video player state machine [36]. The video player has different network behavior (and network-performance requirements) in each of the states. The transitions between states provides insight into whether the Wi-Fi network is providing the throughput that the video player needs. It also provides insight into user behavior while watching video.

layer, we instrument the Android platform's native video player. Like the Wi-Fi subsystem, the video player also has a state machine that represents the process of initializing a video-decoding and video-rendering pipeline, connecting to the video server, maintaining this connection and downloading the video stream, and then finally cleaning up and freeing resources [36]. This state machine is shown in Figure 3.4. Like our instrumentation of the Wi-Fi state machine, we take advantage of various callbacks provided by the Android operating system to inform us of changes to the video states immediately, and we also poll the video player for its current state every 5 seconds in case we miss any state transitions.

By studying the API framework documentation and comparing the state transitions to what is visually shown on screen, we were able to determine the underlying processes being executed in each of the states shown on the diagram. The video subsystem begins in the "Idle" state, where it waits for the application to provide it with a URL of a video file or stream to play. Once the application does this (for example, by providing the URL of a video stream on a server), the player moves to the "Initialized" state but does not attempt to access the resource at the provided URL. It waits until the application calls the `prepare` or `prepareAsync` functions to do this. At this point, the media player needs to download enough metadata about the video to be played in order to initialize the decoding and rendering pipelines. For video content delivered over a Wi-Fi network, this typically involves contacting the server and downloading some metadata about the video, usually with a HTTP request. This, in turn, means that the video player needs to successfully make a connection with the media server via the Wi-Fi network. If it is able to do so successfully, the media player moves to the "Prepared" state. If the media player is unable to do complete this operation due to a Wi-Fi network problem, it remains in the "Preparing" state while the connection is retried, and eventually transitions to the "Error" state if it unsuccessful after multiple tries. A successful transition to the "Prepared" state suggests that a network connection was successful, and usually indicates that subsequent video playback will also be successful. Once the media player is prepared, the application must issue a `start` command to the media player to begin decoding and rendering video content. Once this command is issued, the media player fills its video buffer (by again contacting the server) and playback commences. The media player also provides insight into the buffer state through a callback mechanism, which we also instrument to allow us to monitor the state of the buffer. An empty buffer indicates that video playback will very soon stall, interrupting the user's experience. Once the user indicates that they wish to end their video-playback session (by closing the player, for example), the video player is shut down and moves into the "End" state once all resources are released.

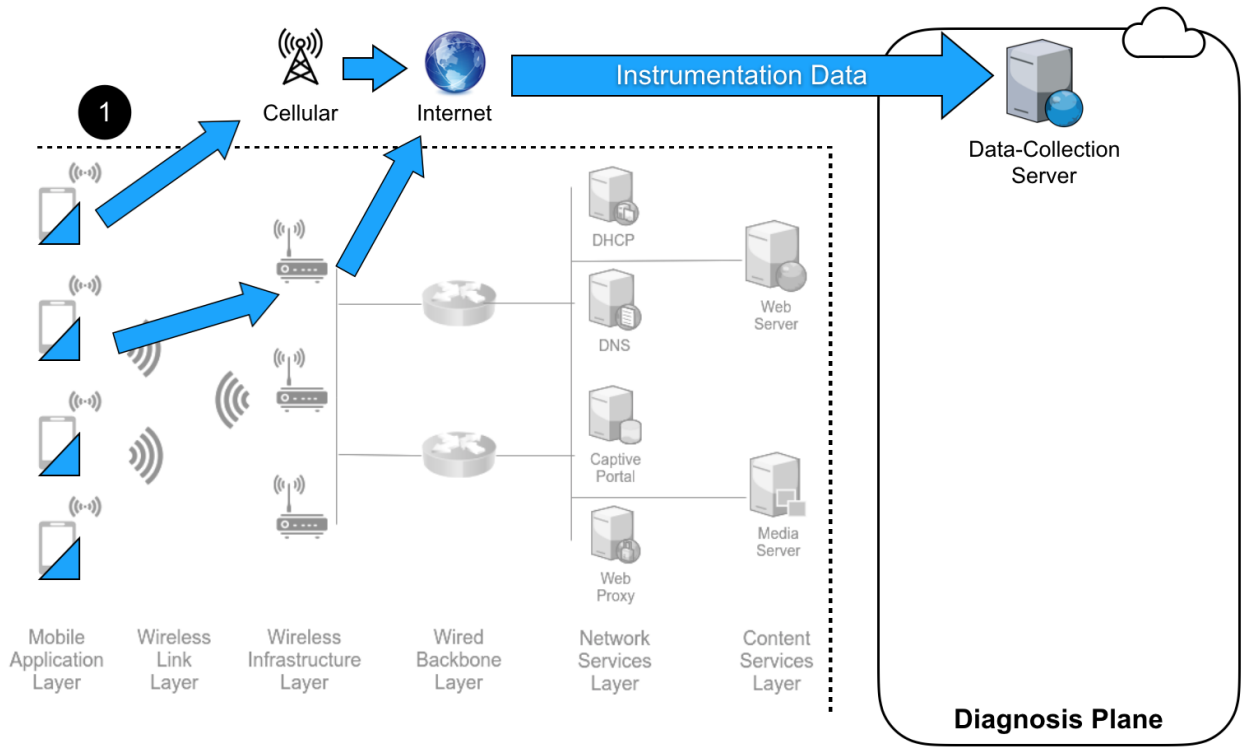

Figure 3.5: A diagram of the high-level architecture of our cloud-based data-collection system. This shows how the end-user devices on production Wi-Fi networks transmit instrumentation data to our data-collection infrastructure. By hosting this infrastructure on the cloud, we do not require any modification of or additions to the on-site production Wi-Fi infrastructure. Our system uses both the Wi-Fi and cellular networks to transport our instrumentation data to provide redundancy if one network is unavailable.

3.3 Data Collection from Mobile Devices

In this section, we aim to discuss our strategy for collecting the aforementioned logs from mobile devices operating in the wild, quickly and reliably, so that we may later process the data and ultimately gain insights from it. Our high-level system architecture is shown in Figure 3.5. As our instrumentation system receives changes to the various monitored devices states and metrics, it divides these logs into sections, creates HTTP requests from these sections, and periodically sends those sections to the data-collection server, as shown in the diagram. Recall that one of our goals was to collect real-world data from users of large-scale, high-density Wi-Fi networks. We may not be able to always have a server on the same LAN segment as the Wi-Fi networks we instrument, so in order to collect this data we need to have an Internet-accessible service that instrumented devices can contact to deliver their logs. Once the server receives a section of logs, it parses those log entries and stores them in a database for further analysis.

Client-side data collection and forwarding to the report-processing server. Figure 3.6 shows the client-side system architecture of our data collection and transport mechanism. Internally, there are separate

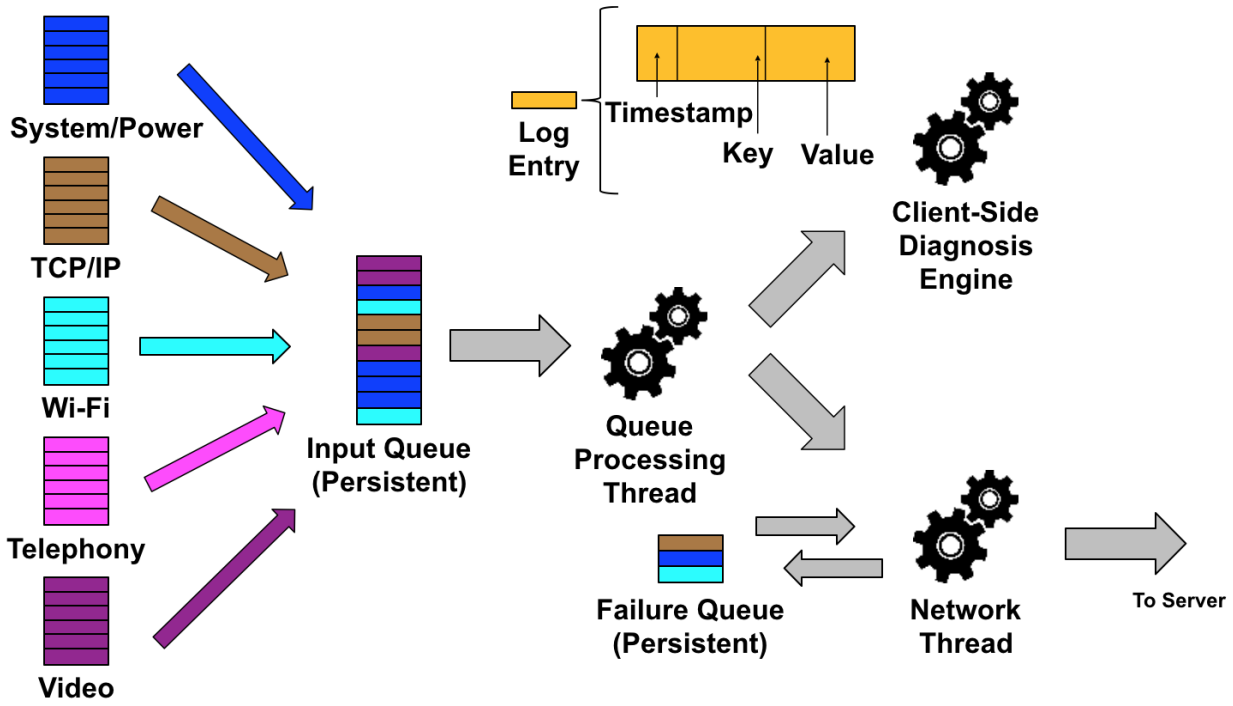

Figure 3.6: A diagram of the data flows within our client-side instrumentation and data-collection system. The various instrumentation points are polled at regular intervals, and the collected data is timestampped and stored in a persistent queue (for robustness against crashes or network failure). Every 30 seconds, the device attempts to send the contents of this queue to the report processing server (as shown in Figure 3.5. If this fails, the data is placed back in to a queue and transmission is attempted later.

Java classes that maintain the state required for the instrumentation described in the previous section. For example, the Wi-Fi instrumentation class contains the operating-system callback functions that are called when scan results are available or the Wi-Fi driver state changes, and also runs multiple thread to poll various operating-system APIs to obtain data that must be polled, like the current link speed.

Each of these classes is instantiated once upon initialization of our instrumentation library (controlled by the application and discussed later in this chapter). Each class maintains its own log of events, which are periodically written into a persistent queue on the device. This persistent queue is stored as a file on the local private storage of the device. We use a persistent queue because the application containing the instrumentation library may be closed or killed by the operating system before it is able to send all of the data that it would otherwise be storing in memory. Also, the amount of data collected may be too large to fit in memory, especially if the server can't be contacted over the network and data needs to be buffered for an extended duration.

At 30-second intervals, a separate thread, the queue-processing thread, reads this queue and gathers the log entries in the input queue into a log segment. A log segment is either the next 1 MB of log entries

in the queue (as determined after serializing them as JSON objects), or the remainder of the queue if the queue contains less than 1 MB of data. The reason for splitting the queue into segments is that our web server, like most web servers, has a limit on the amount of data that it can accept in a single HTTP request. Also, keeping the HTTP request size relatively small increases the chances that the request will succeed. If there is more than 1 MB of queued data, the queue-processing thread will create multiple segments and hand these off to the network thread individually.

The role of the network thread is to receive log segments from the queue-processing thread and attempt to deliver these to the server over whatever Internet connection is available to the device. The network thread packs the next segment, in order, into a HTTP request and sends this via HTTP post to the report-processing server. If this request takes too long (a timeout), or if the server responds with an error, the thread retries the request after an exponential-backoff period up to 3 additional times for the segment. If the network thread fails 4 times to send a segment, the thread puts the segment in the failure queue and moves on to the next segment. Once the network thread has exhausted segments from the input queue, it revisits the failure queue and attempts to re-send those segments to the server. The failure queue is bounded so that long downtime of the report-processing server will not exhaust the disk space on the device.

Devices may not be able to contact the server over the Wi-Fi network in some cases, for example if Internet access is blocked or the Wi-Fi is unavailable. We don't want this to prevent us from capturing Wi-Fi data. Therefore, we allow the device to transmit this data over both Wi-Fi and cellular network connections, any time that such a connection is available. Log entries are transmitted as JSON objects to the web service. The data transmission is secured with HTTPS to prevent this Wi-Fi performance data from being captured by unauthorized third parties when in transit over these public networks.

Once the data arrives at our server, the web service (a Java web service) stores the data in a MongoDB [60] database. This database is co-located with the web service on the data-collection server. We chose MongoDB as the storage engine for this data as it natively provides storage and querying of discrete JSON objects, which matches the format we chose for the transmission of log entries. Our Java server uses the Jetty web server [31] as the underlying HTTP connection manager, which in turn calls our server code for each request. Upon receiving the log entries from a device, our server code splits the request into individual JSON objects, each of which are stored as individual records in a single "raw data" table in MongoDB. As will be discussed later, this table is later processed to group, aggregate, and analyze the data to gain insights from it.

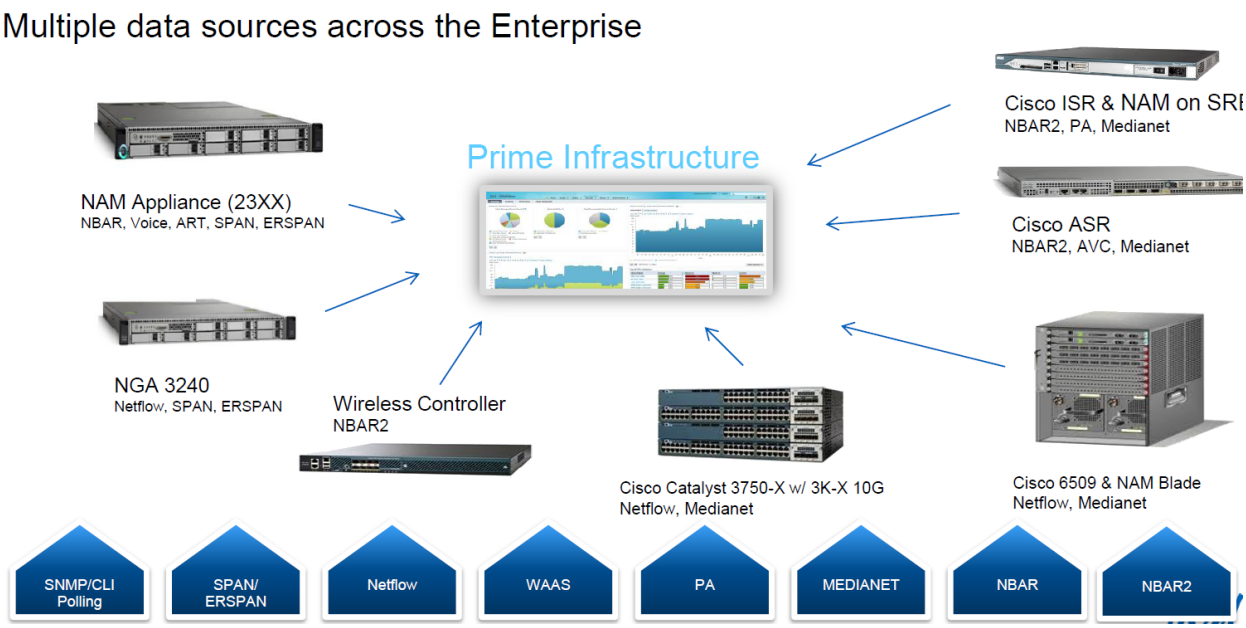

Figure 3.7: A diagram showing the data sources integrated by Cisco PI [63]. Cisco PI collects instrumentation data from multiple Cisco devices within the same local network and exposes this data via a web service. We use this web service to collect the infrastructure's perspective of the Wi-Fi network.

3.4 Infrastructure Instrumentation

This section aims to describe the instrumentation points available in the widely-used Cisco enterprise Wi-Fi platform [72]. We chose to instrument this platform because of its large market share and the centralized availability of various performance metrics that we are able to obtain in a non-intrusive manner (without modifying any Cisco hardware or software). Cisco also rate-limits API calls and updates to prevent third-party systems (such as ours) from overwhelming the CPU on critical infrastructure, which may lead to an outage.

Instrumentation points in the Cisco enterprise Wi-Fi platform. The modern Cisco enterprise Wi-Fi platform is commonly installed alongside Cisco Prime Infrastructure (PI) [20], which provides centralized management and coordination of several otherwise-independent Cisco systems (including Wi-Fi controllers, firewalls, routers and switches, and other network-related hardware). Cisco PI provides the advantage that it also provides a restful API from which one may obtain statistics on access point performance, radio utilization, and even individual Wi-Fi client metrics such as SNR and throughput. Figure 3.7 shows the various infrastructure data sources collected by Cisco PI and made available through APIs.

Cisco provides public documentation [19] around the APIs available from Cisco PI. From this API, we obtain infrastructure-performance data from eight different API calls:

1. **GET Client Summary.** Represents client view with information about the end points. It provides end point information such as MAC address, IP address, username, and status. Using the IP address provided from this API call, we are able to correlate Cisco-collected data with client-side data collected from the device.

2. **GET Client Details.** Represents the detail view of a client. It provides attributes of client device, security information, connected device, traffic and session information. All information is collected in current or last session. This API provides metrics on the length of time that a device has been connected and how much traffic it has exchanged with the AP in its current and previous session.

3. **GET Client Sessions.** Represents detail view of client sessions. It provides device and session related attributes including security, connected device, session time, traffic, etc.. This API provides historical data on all sessions from this device.

4. **GET Client Traffic Information.** Represents clients traffic information collected during the last polling cycle. This includes detailed breakdowns of data transmitted to the client on each Wi-Fi band as well as on the wired network.

5. **GET Client Statistics.** Represents client statistics data collected during the last polling cycle. The data represented here are the counters retrieved from controllers. The counters only get reset in new sessions. This API provides data on the number of packets transmitted and received for this client over the radio, the number of packet retransmissions, and the number of packets lost.

6. **GET Lightweight AP Radio Details.** Represents detail information of a radio interfaces on lightweight access point. A "lightweight" access point, in Cisco terminology, is an access point that requires a controller to function (cannot operate in a standalone mode). This API call returns performance data aggregated on a per-radio basis, including radio configuration (current channel, transmit power), administrative status (enabled/disabled), alarm status, and other "metadata" about the radio.

7. **GET Radio Interface Statistics.** Represents the latest statistics information for radio interfaces of lightweight wireless access points collected from WLAN controllers. The data provided via this API includes the number of clients connected to each radio, along with four alarms with configurable thresholds. Those alarms are: (i) coverage profile, raised if the radio detects that clients are attempting to access the Wi-Fi network in an area where there is a coverage gap; (ii) interference profile, raised if the radio detects an abnormal amount of 802.11 interference on the channel it is using; (iii) load profile, raised if the radio detects that it is overloaded, and (iv) noise profile, raised if the noise floor (non-802.11 interference) is too high.

8. **GET Radio Interface 802.11 Counters.** Represents the latest 802.11 counters for radio interfaces of lightweight wireless access points collected from controllers. This API provides counters for the number of packets that fall into various states, such as successful transmissions, retransmissions, multicast packets, ACK failures, and RTS/CTS failures.

Unlike our Android instrumentation, Cisco devices themselves do not run any instrumentation-related code. Instead, our report-processing server polls the Cisco PI APIs directly at 2-second intervals. Each time, all eight of the aforementioned APIs are polled for each client that the report-processing server has seen (based on the IP address included in the client-side data). The report-processing server uses this IP address to map infrastructure-side data to each individual device, so that all Cisco PI data relevant to that device is stored along side the client-side data in the same series of log entries. We assume the clocks on both the server and the clients are synchronized, as the server needs to generate timestamps for the infrastructure-side log entries independent of the Android device's timestamps. As with the client-side data, these infrastructure-side log entries are stored in the MongoDB database.

A drawback to the infrastructure-side data is the frequency of updates that it can provide. By default, Cisco PI gathers updates on many of the metrics described above at 5-minute, 15-minute, or 1-hour intervals. These are safe intervals meant to protect the system from high CPU overheads caused by gathering and processing statistics (note that the same hardware used to gather this data is also simultaneously processing traffic on the wired or Wi-Fi networks). We attempted to tweak these intervals to the minimum possible in the system. Unfortunately, the minimum interval we were able to set across any of the APIs was 1 minute. Some APIs, including those with detailed client and radio packet counters, only updated a roughly 5-minute intervals. While these intervals may be sufficient for reviewing the overall operational status of the network or diagnosing widespread, persistent problems, they are not granular enough to provide detailed insight into short-lived local problems in the network such as the ones we seek to diagnose.

3.5 Non-Intrusive Deployment Strategy

As stated in our goals, we aim to deploy our instrumentation and data-collection system in a non-intrusive manner. We do this by packaging our instrumentation framework as a library that can be integrated with any Android mobile application. To include this with an app, the app developer need only add the library to their software project, usually as an Android archive (AAR) file. The library defines a public-facing interface to allow the application to control which instrumentation is installed at runtime as well as the frequency of data collection and reporting.

We intend for this library to be included in a mobile application commonly used on the Wi-Fi network(s) being instrumented. For example, the live-video streaming application used at stadiums and arenas are typically bundled within a mobile app associated with the team or venue. Other applications could similarly make use of this library, for example a free video-streaming service used in airports or an educational video-on-demand service used in schools. As users are downloading an application that they would otherwise download to take advantage of a service provided by the facility in which the Wi-Fi network is located, and our instrumentation and data collection is simply "piggybacking" on that application, we consider this a non-intrusive method of deploying our framework in real-world large-scale high-density Wi-Fi networks.

For the production live-streaming-video application that we studied, the library is integrated as an external source tree and the background instrumentation service is registered on application launch. The mobile application initializes the data-reporting system when the user accesses the live-streaming or instant-replay video features of the application. Once initialized, the application registers a number of callback functions for various system information, such as driver state. When a video begins playback, the application registers a callback to retrieve information about the state of the media player, as described above. When the media player is about to be destroyed because the user is done watching video, media-player callbacks are de-registered and the media player is no longer polled. Additionally, the data-reporting system installs callbacks with the operating system to be notified asynchronously of state changes (when possible), such as Wi-Fi associations and Wi-Fi scan results.

Chapter 4

Problem Detection

This chapter seeks to answer the question of how our system can detect when a problem has occurred, and also how can we do so quickly. The initial part of this chapter begins by describing how we can gain understanding of when a technical problem has occurred. The latter part of the chapter discusses when the technical problem begins having a negative impact on the user experience.

To provide some context for the Wi-Fi-network-performance requirements for live-video streaming, we start with a detailed discussion the behavior of a common video-streaming protocol used to deliver video to mobile devices. This protocol is used by our live-video-streaming application, and it (or variants of it) are used by nearly every major video-streaming service for mobile devices today. We then discuss the relationship between this video streaming protocol and the states and events within the Android video player, and when technical problems become user-visible. Finally, we detail how we do this problem detection through analysis of the instrumentation logs we collect.

We then move on to a description of how the streaming-protocol's operation maps to the video-player states we discussed in the previous chapter. This allows us to identify which states require network communication, making it more likely that an error or delay is due to a Wi-Fi performance problem while in that state (and the device is communicating via the Wi-Fi network).

Finally, based on our understanding of the video-streaming protocol and the relationship between the protocol and video-player states, we introduce our problem-detection algorithm. This algorithm monitors the transitions between video-player states and the length of time that the player is in these states to determine whether a problem has occurred. The length of time that the player needs to remain in a state to trigger the detector are parameters of our system, and we describe how we set those parameter in the next chapter.

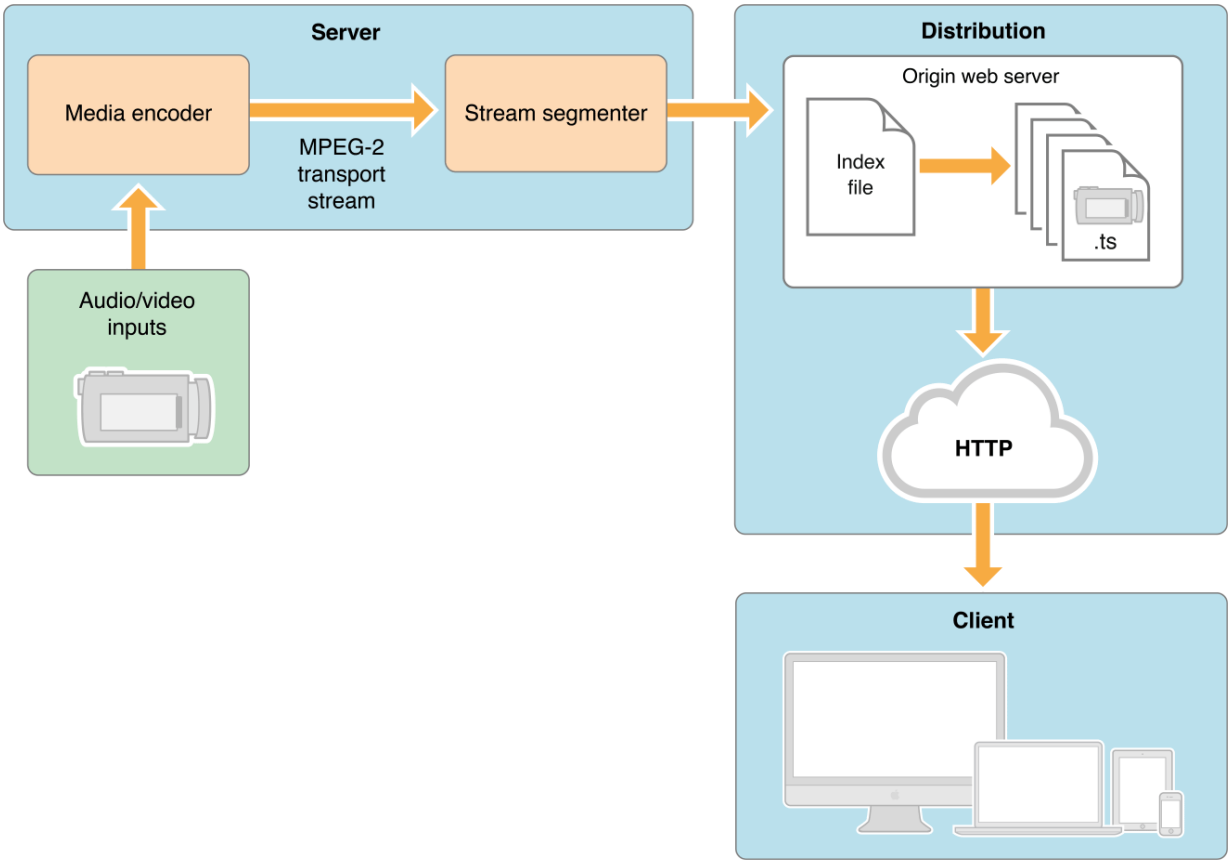

Figure 4.1: The architecture of a typical HTTP Live Streaming system [6].

4.1 Mobile Video-Streaming Protocols

Apple's HTTP Live Streaming, or HLS, [6] was one of the first chunk-download video-streaming protocols developed and is still in wide use today. It has spawned numerous derivative protocols, such as MPEG DASH [30], and has also been used as the basis for a number of competing protocols. The distinguishing behavior of this class of protocols is that the video stream is split into discrete chunks that can be played independently of each other. This means that each chunk contains all of the metadata needed for the video player to understand how to play the video within that chunk.

Server-side operation of HLS. Figure 4.1 shows the architecture of a typical HTTP Live Streaming system. The audio/video inputs are some form of raw audio-video stream, such as a HDMI connection into a computer or server. The server reads that raw stream and passes it through a video encoder, which outputs a compressed video stream in a MPEG-TS container. The video compression typically uses the common H.264 codec for video compression, and the AAC or MP3 codec for audio compression. Once the compressed audio and video streams are encoded, the server multiplexes these streams in a MPEG-TS

container. This container is essentially a file format that begins with some metadata about the stream (which codecs are used, how many streams are present, etc.) and then follows this with interleaved audio and video data, with timestamps that indicate to the media player when each audio or video frame should be played.

The MPEG-TS file format has the property that it can be spliced into two separate files along a packet boundary, and both files are playable as independent video files. The inverse is also true. Simple bit-wise concatenation of two MPEG-TS files will produce one longer MPEG-TS file with no distortion or interruption as playback moves between the two files (the inverse of splicing). HLS takes advantage of this property to ensure smooth playback of video even though the stream itself is distributed in discrete chunks.

Once a MPEG-TS stream is generated, it is passed to a stream segmenter which breaks the stream into discrete files known as chunks. These chunks are typically given names with some sequential numeric identifier (e.g. `chunk_001.ts`, `chunk_002.ts`, and so on). The length of the chunk is determined by the acceptable latency for the stream. Smaller chunks are required for lower latency streams, since a chunk of length T seconds requires that the server buffer video for at least T seconds before making the chunk available to download. On the other hand, smaller chunks require more frequent downloads, generating more protocol overhead (from individual requests) and also more contention for uplink airtime in Wi-Fi networks. Typical values for T are 30 seconds for on-demand videos or streams where latency is not a factor, 10 seconds for cases where latency is a moderate concern, and as low as 3 seconds where latency is the primary concern for stream performance.

In addition to the individual chunks, the stream segmenter also has the responsibility of generating the index file for the stream. The index file typically resides at a well-known URL (which is provided to the media player) and contains the URLs of each of the individual chunks of the stream. For live streams (with indeterminate length), the index file will always contain just the latest N chunks of the video. System operators choose N based on latency requirements and also how much buffering is required to deliver an uninterrupted streaming experience. Typical values for N range between 4 and 10 chunks.

The index file and the individual chunks are then made available to video-streaming clients over HTTP. This can be as simple as a standard HTTP file server, for example Nginx or Apache. Many large-scale stream providers such as Akamai use sophisticated caching of the index file and chunks on multiple edge caches to improve performance [57]. Our live-video streaming system also takes advantage of this property to cache cloud-originated video chunks on a server in a LAN with in-venue Wi-Fi devices to improve latency and reduce the load on venue Internet links [53].

Client-side operation of HLS. To play back a HLS stream, the client begins by downloading the index

file from the web server via a well-known URL. The list of chunks in the index file is then used to fill a client-side video buffer, usually several seconds in size. The size of this buffer is typically set using metadata found in the index file, or if this is not present the media player may set a buffer size based on the number of chunks found in the file. For live-video streaming application under study, the buffer size is 12 seconds, made up of 4 chunks that are each 3 seconds in length.

At the start of playback, the client attempts to fill its buffer and does not begin playback until the buffer is full. This is to ensure that, once playback begins, if there is a temporary interruption in download performance, the buffer will provide uninterrupted playback to the user as long as the interruption does not last longer than the data remaining in the buffer.

Once video playback begins, the goal of the media player is to keep its buffer full. Once it has downloaded the latest chunk of the stream, it waits for a chunk duration before downloading the next chunk in sequence (to give the server time to produce the next chunk). If an error occurs while downloading a chunk or if the rate of download falls behind the rate of playback, the player will continue to attempt to download chunks in an effort to re-fill the buffer. If the player continues to fall behind due to an inability to download chunks fast enough to keep up with the rate of playback, the buffer will eventually underflow. At this point, the player can no longer hide the network-performance issue from the user (as there is no video remaining to play), and typically a "Buffering..." message or loading animation is shown to the user. This is the point at which Wi-Fi network performance problems become user-visible errors that will begin to have an impact on the user experience.

Relationship between chunk length and bandwidth consumption. The above discussion of the HLS protocol describes chunks in terms of their length in time, which is the wall-clock time required to play the chunk end-to-end at its recorded frame rate. A chunk can be encoded at different bit-rates, which means the encoder is free to use more or less bits per second of video at higher or lower bit-rates. The bit-rate of a stream provides a way to relate chunk length to the actual bandwidth required to download the stream. Typical bit rates for video streams are 500 kilobits per second (Kbps) for standard-definition streams, and 2 megabits per second (Mbps) for high-definition streams.

4.2 Mapping Protocol Behavior to Media-Player States

Now that we have reviewed the operation of chunk-download protocols like HLS, we describe the mapping from the protocol operation to different states and events in the Android media player. Chapter 3 described the media player and the set of states and events that occur during its operation. We briefly review these below and discuss how they map to the HLS protocol operation.

1. **Idle.** The media player is idle, no protocol operation has begun.
2. **Initialized.** The application has provided a URL of the HLS index file for the stream to the media player.
3. **Preparing.** The media player is attempting to download the index file using the provided URL. It may also attempt to download one chunk of the video to obtain metadata if the metadata contained in the index file is insufficient.
4. **Prepared.** The media player has downloaded the index file and enough chunk data to configure the decoding pipeline and is ready to begin playback.
5. **Started.** The media player has downloaded all of the video chunks needed to fill its buffer and has started playback.
6. **Playback Completed and End.** The player has reached the end of the stream (indicated by no more chunks listed in the index file), or the user has requested that the video player close.
7. **Error.** The video player was unable to understand the metadata contained in the index file or chunk files, there was an error in the encoding of the video, or the application received too many server errors while attempting to play video.

4.3 Problem Detection via Log Analysis

Before discussing our problem detection approach, we introduce some terminology to accurately describe video playback and the possible conditions that can occur during video playback, as follows.

- **Video Playback Session (VPS).** A video playback session is defined as the duration between the user requesting a video stream and the user closing the video player. The user starts a video playback session by tapping a button in the mobile app that launches the video player with the URL of the requested stream. This launch is indicated by a "initialized" event in our logs. The session ends once the user exits the video player by tapping the back button on the device, which is indicated by a "finalized" event in our logs. About 10% of video playback sessions in our logs do not have both initialized and finalized events, possibly due to the app being closed or forcibly terminated before the system has been able to transfer the logs to the server.
- **Playback Position Counter (PPC).** This is a counter incremented by the video player for each millisecond that video is played.

- **Initial Buffering Time (IBT).** This is the duration between the user requesting a stream and the stream beginning playback. The request is indicated by a "initialized" event in our logs. The beginning of playback is indicated by the PPC exceeding zero, which corresponds with the video player moving into the "Started" state. Note that the user is waiting for playback to begin during this period, and is typically viewing a black screen or a loading animation.

- **Initial Buffering Cancellation (IBC).** A VPS is considered to have been canceled during initial buffering if the session transitions to the "End" state during the IBT. This event indicates that the user requested playback, but closed the player (using the back button) before playback began.

- **In-Stream Buffering (ISB).** In-stream buffering is indicated by the PPC falling behind the log-entry timestamps by more than 1 second during a stream. This indicates that the there was a user-visible interruption in video playback that caused the video to fall behind.

For the purposes of problem detection, we consider a problem to be an event that has a negative impact on user experience due to a systems-level issue. We focus on two metrics described above as primary indicators of problems: lengthy IBT intervals and ISB events. Both of these events are heavily dependent upon the performance of the network to deliver data in a timely manner in order to keep the IBT short and prevent ISB events. The metrics listed above other than IBT and ISB are secondary indicators of system state that may be related to faults. We discuss how some of those metrics contribute to our understanding of why faults occurred and how they correlate with the primary metrics later in this dissertation.

To detect the signatures of long IBT times and ISB events, we developed an algorithm that scans the sequence of log entries generated by our instrumentation framework, in timestamp order, and outputs timestamps of when problems occurred, and also which class (IBT or ISB). Being a streaming algorithm, our approach can be used in both online and offline modes to identify problems in near real time, or to determine when problems occurred in historical traces.

Figure 4.2 provides an overview of the operation of our problem-detection algorithm. Our algorithm begins by assuming that the video player is in the "Idle" state and searches for a state change to "Initialized". Until it sees the first "Initialized" state, it ignores any other state changes (which could happen, for example, if the instrumentation was started in the midst of a VPS, or data is lost. Once the problem detector sees the "Initialized" state, it records the timestamp at which this state was entered for future use.

The problem detector then continues examining log entries, looking for either an entry indicating that the PPC is greater than zero or the media player has transitioned to the "Finalized" state. Note that

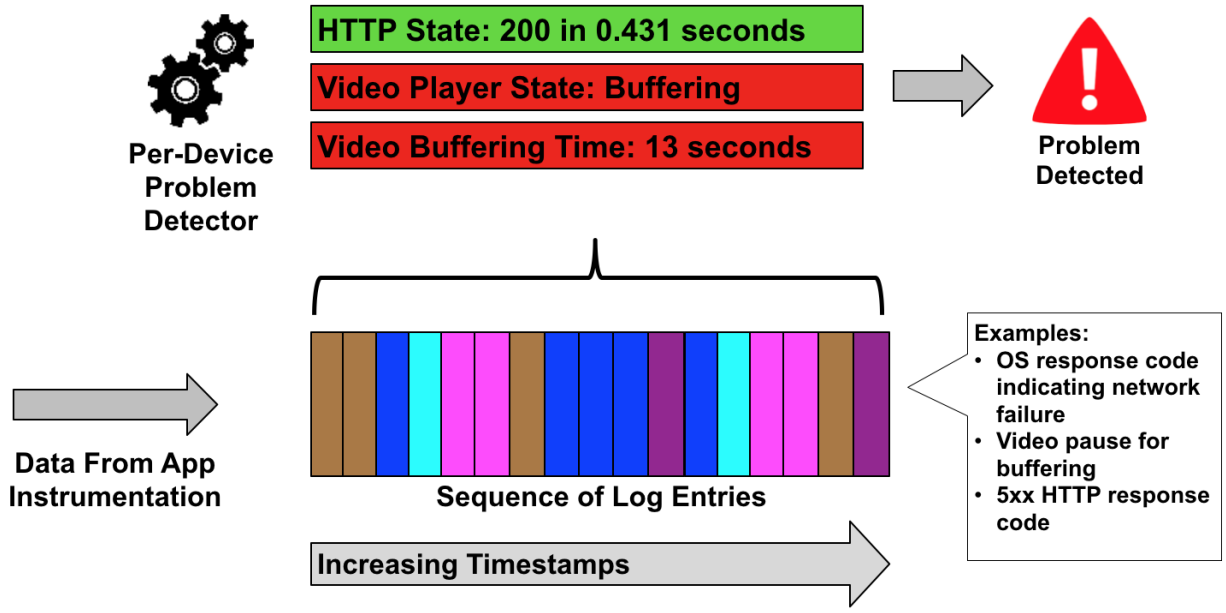

Figure 4.2: A diagram describing the operation of our problem-detection system. The problem detector, running on the end-user device, continuously monitors the sequence of log entries produced by our instrumentation system. When the detector reads either an entry explicitly indicating a problem or sequence of log entries that indicates a problem, it signals a problem alert to the problem-diagnosis system.

we do not use the state transition to the "Started" state, as we found through experimentation that the media player does not reliably emit this state, and the PPC value is therefore a more reliable measure of when playback has begun. At this point, the problem detector compares the recorded "Initialized" timestamp to the timestamp when the PPC exceeded zero or the media player moved into the "End" state. If the difference between these timestamps surpasses a threshold, the problem detector outputs an "IBT" problem at the current timestamp. If the media player has moved into the "End" state, the problem detector returns to looking for a subsequent "Initialized" state.

Assuming the media player did not move into the "End" state, the problem detector then shifts to searching for ISB events. While the media player emits an event when buffering begins, we found experimentally that these events are not reliably emitted from the Android media player. Therefore, we developed a heuristic to detect when video playback has fallen behind real time enough that buffering has likely occurred. We detect an ISB event by comparing the PPC (the number of milliseconds of video played back) against the wall-clock time elapsed since the PPC first exceeded zero. If this difference exceeds one second, the problem detector notes the current timestamp as the start of an ISB event and looks for the next event where the PPC increases. Once the PPC again increases, the problem detector assumes buffering has ended (as video playback has made progress) and compares the current timestamp against

the timestamp recorded at the start of the ISB. If the difference between these timestamps surpasses a threshold, the problem detector outputs an "ISB" problem at the current timestamp. The detector also uses the current timestamp as the basis for future ISB detection. This process of ISB detection continues until the video player reaches the "End" state, or another "Initialized" state is seen.

Open questions. The above description of the problem detector leaves open the questions of how to set the two thresholds mentioned in the description, one for the length of the IBT before outputting an IBT problem, and one for the length of the ISB before outputting an ISB problem. Users understand that it takes some time for a video stream to buffer at the beginning (the IBT), and will tolerate IBTs up to some length. Similarly, users understand that occasionally network issues will cause buffering to occur while they are watching video, and they tolerate this up to a certain point. The next chapter will explore these thresholds in more detail and discuss how to set them, based on our study of real-world data on user-behavior.

Chapter 5

Characterization of Video-Playback Errors and User Behavior

As mentioned at the end of the previous chapter, our problem detection approach leaves two key thresholds undefined. The first is the length of time that the IBT must exceed before it is flagged as a problem. The second is the length of time that an ISB event must exceed before it is flagged as a problem. This chapter seeks to answer both of these questions through a study of real-world user behavior with the YinzCam video-streaming application on real-world, large-scale high-density Wi-Fi networks.

5.1 User Reactions to Video-Streaming Problems

To begin our study of how users react to video-streaming delays, and when they become problems, we introduce two additional events that build on the video-playback events and definitions introduced in the previous chapter.

1. **Initial Buffering Cancellation (IBC).** A VPS is considered to have been canceled during initial buffering if the session is finalized during the IBT. This event indicates that the user requested playback, but closed the player (using the back button) before playback began.
2. **In-Stream Buffering Cancellation (ISBC).** A VPS is considered to have been canceled during in-stream buffering if the session is finalized during ISB. This occurs when the user closes the player while the session is experiencing ISB.

These metrics were chosen because these events indicate that the user canceled video playback after expressing a desire to watch video but while the system was not able to play video due to a system-related

issue. In both cases, a loading spinner indicative of buffering is shown to the user during this time (in place of video), although the issue may be related to buffering delays or some other problem (e.g. no response from the server). The act of canceling playback during this time strongly suggests that the user decided to no longer wait for playback to begin or continue, thus resulting in a negative user experience.

5.1.1 Production Data-Set and Traces

Using our instrumentation and data-collection system, we collected over 2 years of data from 35 mobile applications used in 25 sports venues. The analysis below includes data from 800 days of our data set, ranging from August 8, 2013 to November 26, 2015. This data set has 31,954,638 log entries collected from 292,037 unique devices.

Our data-set is large because it has been collected from many different sources over a long period of time. However, this means there are multiple dimensions of the data that we will consider later in our analysis. First, our data-set is an aggregate of data collected across many sporting events, from 25 sports venues of different sizes (20,000-70,000 fans). When viewed as a whole, the data-set gives insight into the Wi-Fi performance at an average sporting event, and how Wi-Fi performance varies across venues and over time.

The data was collected whenever a user accessed the in-venue section of the team mobile app. This section is accessible outside the venue, and once the user attempts to play video, the app will display a message to the user informing them that they must be inside the venue. This means that some of our data may be collected outside the venue, which is not useful for our analysis (which is focused on in-venue usage).

To guarantee that the data we analyze is only from inside the venue and over Wi-Fi, we examined the log entries for each device, within each app over the course of a day, and verified that the device was able to see a known in-venue Wi-Fi network (as reported via the "Nearby SSID count and list" metric). If a device was not within range of one of the known in-venue Wi-Fi networks, its data was excluded from the data-set for that app during that day. Furthermore, we excluded from the data-set those app-days where no sporting event was played. The resulting filtered data set included data from 9,034 unique devices across 234 app-days.

5.2 Thresholds for Problem Detection and Diagnosis Latency

By analyzing our production data-set, we were able to gain insight into how IBT and ISB events relate to user behavior. By understanding user behavior, specifically when and why users disengage, we are

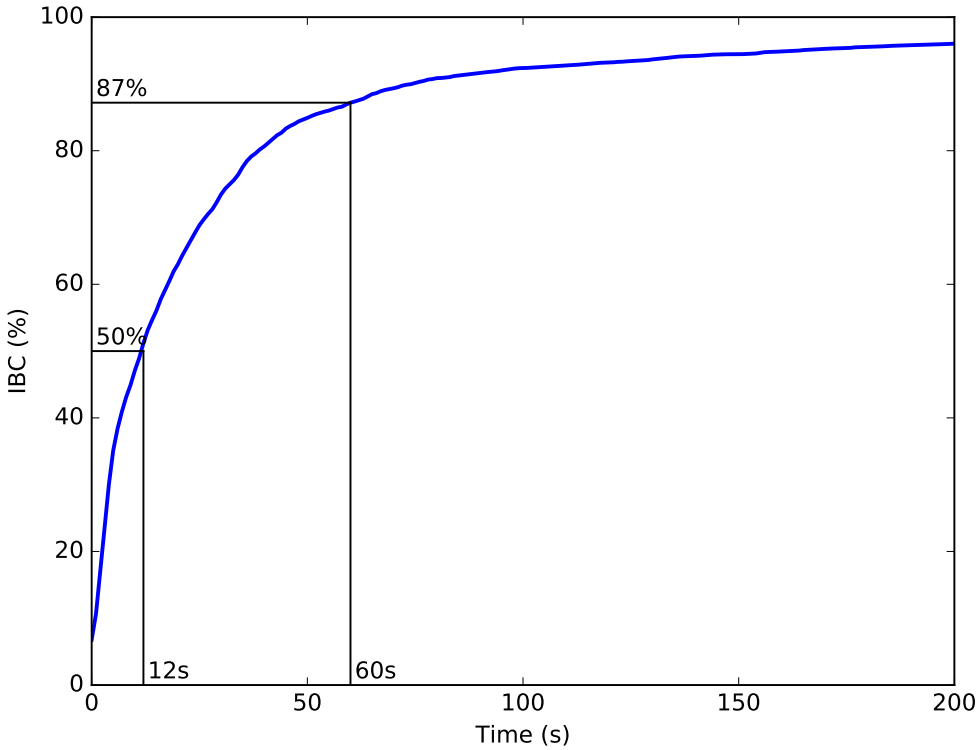

Figure 5.1: The CDF of IBCs as the IBT increases. The x-axis indicates the number of seconds from the beginning of playback (the length of the IBT). For clarity, the CDF does not show the upper 5% of the range beyond 200 seconds. On average, users cancel their sessions about 50% of the time once the IBT reaches 12 seconds. This sets the threshold for a user-visible error at 12 seconds of initial buffering time.

able to define which errors in video streaming cause an unsatisfactory user experience. Once we know the characteristics of an unsatisfactory user experience, we can design systems to avoid the problems that cause these poor experiences. Furthermore, if we can detect when an unsatisfactory user experience, we can attempt to diagnose the problem and automatically mitigate it, or if that is not possible, compensate the user for it in other ways.

We describe here our findings on user behavior while using our in-venue video streaming application, and attempt to quantify the unsatisfactory user experience. Specifically, we focus on how playback errors, such as long IBTs and ISB events, affect user behavior. This allows us to discover the thresholds where most users begin to disengage from the application due to these errors, thus informing the thresholds that we set in our problem detector.

5.2.1 Characterization of User Behavior

Our first insight, drawn from Figure 5.1, shows that about half of all video sessions are canceled once the IBT reaches 12 seconds. In these cases, users are waiting for video playback to begin, and are being shown

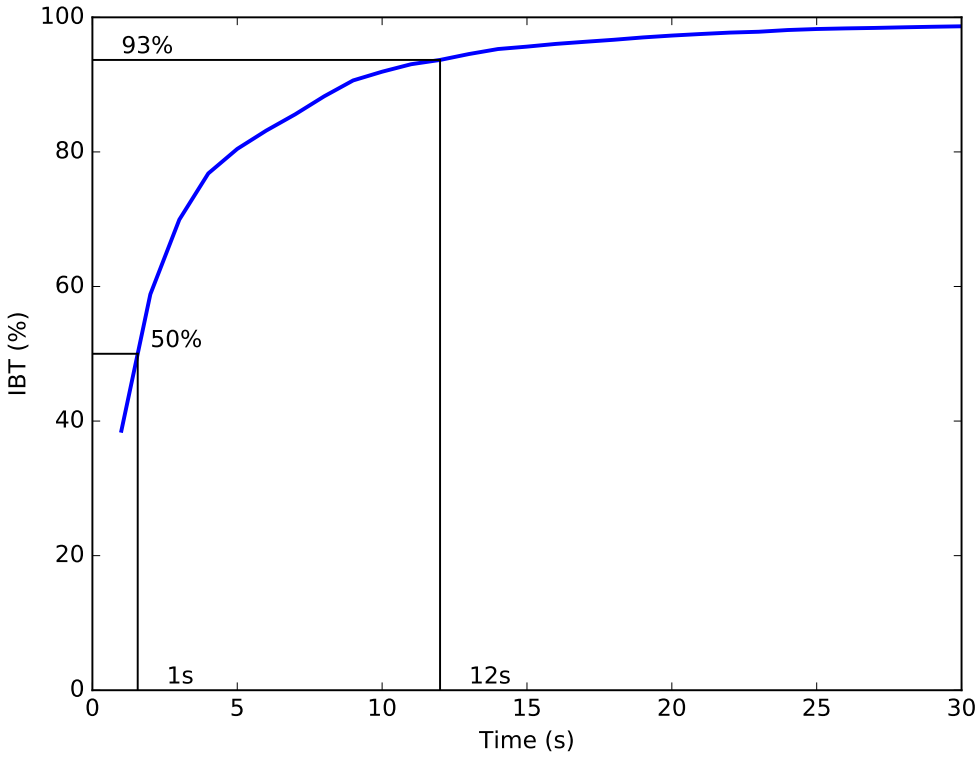

Figure 5.2: The CDF of IBT durations for VPSes that begin playback, showing the proportion of initial buffering phases that complete within the amount of time shown on the x-axis. For clarity, the CDF does not show the upper 2% of the range beyond 30 seconds. About half of all VPSes that begin playback do so within 2 seconds, and 93% begin within 12 seconds. This shows that only 7% of the video streams that we studied took longer than our 12-second user-visible error threshold.

a loading spinner indicating that the stream is buffering. Because most users disengage after 12 seconds of buffering, it's important to ensure that most streams complete buffering within 12 seconds. Of course, shorter buffering times are always better, as even slight delays will cause some users to disengage.

Figure 5.2 shows the CDF of IBT durations across all VPSes that complete the initial buffering phase. VPSes that were canceled before initial buffering was completed were not included in this figure. Our analysis shows that 50% of these sessions complete buffering within two seconds, and 93% within 12 seconds. Only 3% of VPSes complete after 20 seconds, suggesting that once a video has been buffering for more than 20 seconds, it's very unlikely that it will ever complete. At this point, the user is forced to close the video player manually and try again.

Unfortunately, users that cancel streams often will not try again, even though subsequent attempts may be successful. We grouped VPSes resulting in an IBC into clusters by considering two to be in the same cluster if they occurred within 10 seconds of each other. We found that 65% of these clusters have only one VPS, meaning that 65% of the time, users disengaged after only a single failed attempt. A further

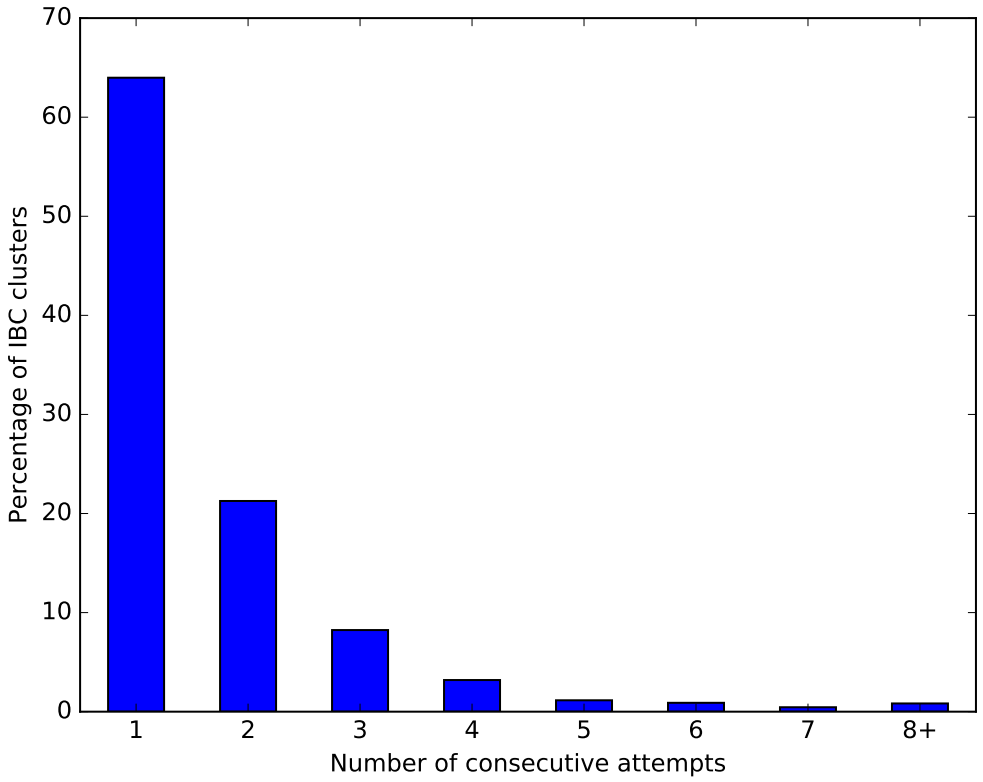

Figure 5.3: The percentage of users disengaging from the live-video-streaming application as the number of consecutive attempts at video playback increases. Most users (65%) disengage after just the first attempt, while less than 5% of users will attempt video playback more than 3 times before giving up. This shows that it is important to address the problem immediately after the first failed attempt at playback.

20% disengaged after two attempts, and 10% after 3 attempts. This result highlights the importance of the video working the first time for the user.

We also explored the number of cancellations occurring during in-stream buffering events (ISBs), in contrast with IBCs. On average, by the time that a session has experienced ISB for 6 seconds, half of all users will have closed it. This is in contrast to the 50% mark for IBC, which occurred at 12 seconds. Thus, while ISBs are rarer than IBCs, it's more important that ISBs be shorter so that users do not disengage.

5.3 Discussion

The above results suggest appropriate thresholds to set during problem detection for IBT length and ISB event length. Given that 50% of users will disengage once IBT length rises to 12 seconds, our problem detector flags any IBT longer than 12 seconds as a problem. Similarly, given that 50% of users will disengage once ISB event length exceeds 6 seconds, our problem detector flags any ISB event longer than 6 seconds as a problem.

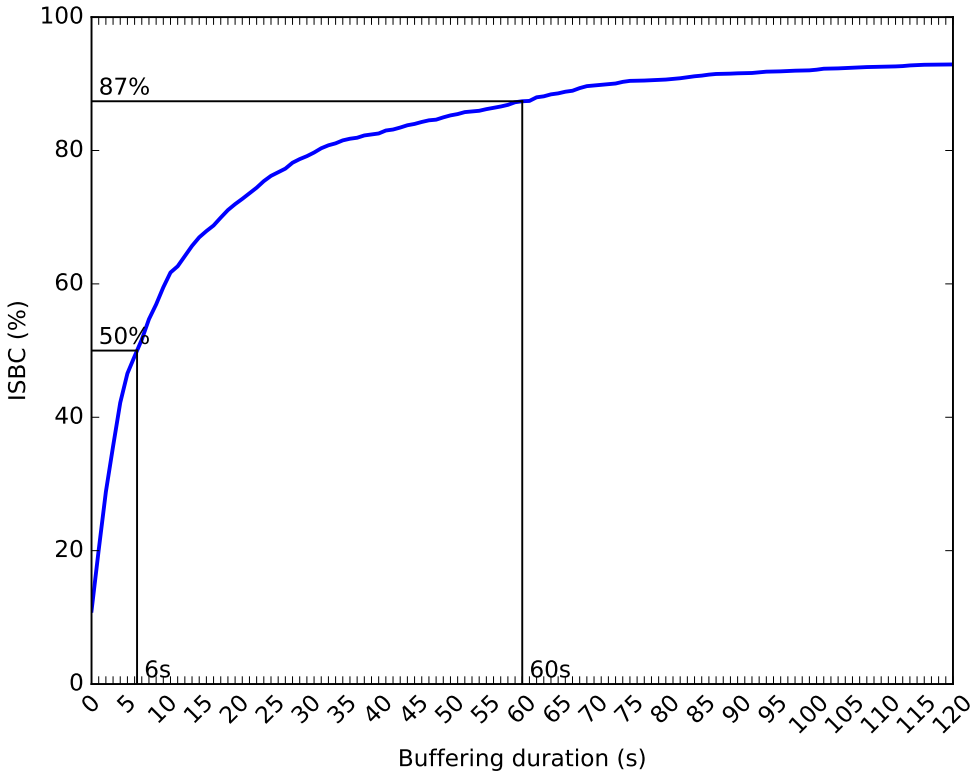

Figure 5.4: The CDF of ISBCs as the ISB delay increases. For clarity, the CDF does not show the upper 8% of the range beyond 120s. On average, users cancel their sessions about 50% of the time once the ISB delay reaches 6 seconds. This is in contrast to the 50% threshold for IBT, at 12 seconds. This shows that users are more tolerant of initial buffering delays than in-stream buffering delays, and we should set our threshold for a user-visible problem at 6 seconds of in-stream buffering.

Chapter 6

Problem Diagnosis and Root-Cause Analysis

In this chapter, we describe the development of our diagnostic approach, including our study of relationships between problems and Wi-Fi performance metrics, our fault model, and the development, and our algorithm to diagnose faults within that fault model based on our instrumentation data.

We begin with a study of the relationship between the data collected via our instrumentation framework and the performance problems that we see in the wild. This is a starting point to begin understanding how our measurements of the Wi-Fi network correlate with user-visible problems. Next, we present our fault model, which is the set of faults that occur in real-world Wi-Fi deployments, which we have chosen to study. We then discuss the patterns that dominate for different faults, based on the analysis of our instrumentation data, and identify a set of key indicators for each fault. This leads to a rule-based algorithm for diagnosing faults, based on automated matching of instrumentation data against the patterns identified through our analysis. Finally, we discuss ways to automatically set the various parameters and thresholds within our diagnostic algorithm to make our approach applicable across a wide variety of Wi-Fi networks.

6.1 Relationship between Wi-Fi Performance Data and Detected Problems

This section presents our study of how our Wi-Fi performance data correlates with the problems that we have seen in the wild. While this analysis doesn't provide conclusive evidence of causation, it suggests what factors may be related to Wi-Fi problems and are therefore likely indicators of problems and and their root causes.

To give an idea of the range and frequency of RSSI values measured by devices running our application, we plotted the distribution of these values according to the time that devices spent at each level in figure 6.1. While the device is connected to an AP, the device logs an update each time that the measured RSSI value changes. The time between consecutive measurements was calculated and applied to the earlier measurement. These times were then summed across all RSSI values seen to create the distribution shown. The distribution is normal with a mean of -63.6 dBm and a standard deviation of 6.7 dBm. Most devices, therefore, measure dBm values within about -57 to -70 dBm. Values below -70 dBm may indicate a network-coverage problem.

To see how RSSI relates to errors such as IBC and ISB, we computed the proportion of these errors occurring at each measured RSSI level relative to the number of streams (IBC) or amount of playback time (ISB). The results are shown in Figures 6.2 and 6.3. In figure 6.2, the RSSI value used was the last measurement taken by the device when the VPS was initialized. For clarity, RSSI values containing fewer than 30 samples were omitted. There is a clear trend toward a higher proportion of IBC events as RSSI decreases. In figure 6.3, RSSI values accounting for fewer than 13.8 hours of playback time were omitted for clarity. We found that there is a weak trend toward a higher proportion of ISB time as RSSI decreases.

Both results show a correlation between increased errors and lower (weaker) RSSI, but the correlation

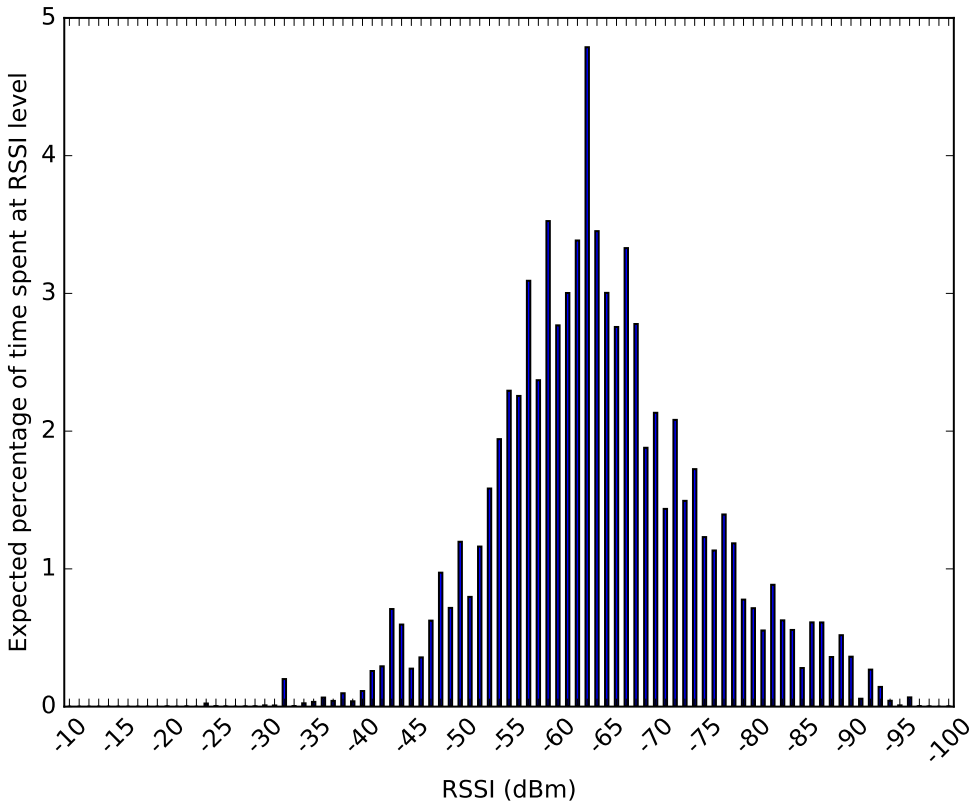

Figure 6.1: The distribution of time that devices spent at client-measured RSSI levels during video playback. This shows that most RSSI values during playback are at -70 dBm or higher. The sharp dropoff around -70 dBm may indicate that video playback becomes problematic when RSSI drops below -70 dBm.

with IBC was more pronounced than that of ISB. The likely cause of this difference is that the RSSI is proportional to the link speed (as higher signal power for a constant noise power provides higher channel bandwidth), and is therefore indicative of the bandwidth available to the device. If this bandwidth is below the bandwidth required for video playback, the initial buffering phase will take a long time to complete and is therefore more likely to be canceled by the user.

6.2 Fault Model

Through our study of related academic literature, the analysis of our collected data, and our understanding of the architecture of Wi-Fi networks, we are able to categorize faults by the Wi-Fi layers in which they occur. Table 2.1 describes some representative faults in each of these layers. There are many more faults that could be listed under each of these layers, but our goal is to narrow down the wide variety of faults that may occur to only those which occur in the mobile-application, wireless-link, and wireless-infrastructure layers, which are the focus of this thesis.

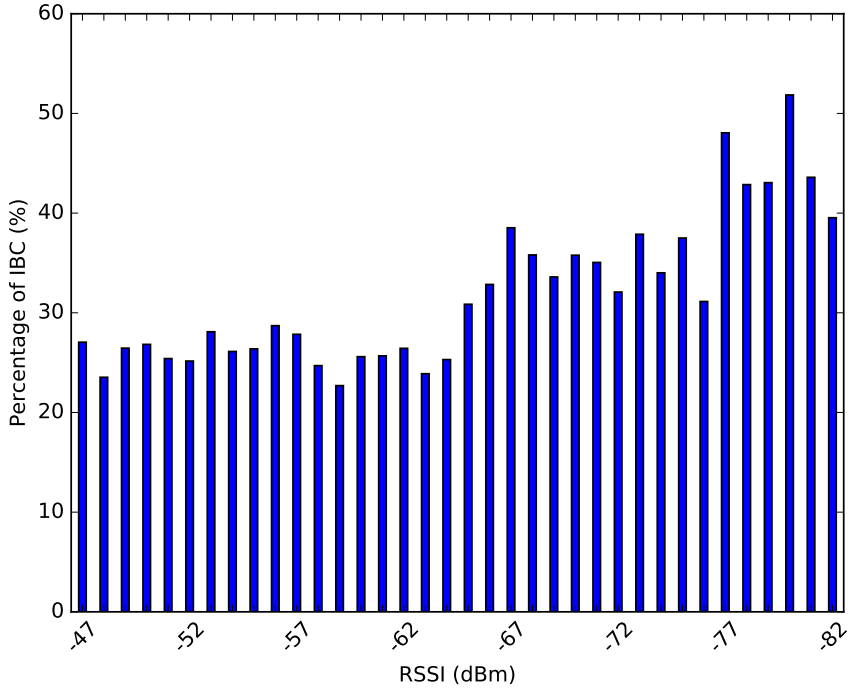

Figure 6.2: The number of IBC events occurring during each client-measured RSSI level. The slight upward trend toward the right side of the chart indicates that lower RSSI measurements correlate with more IBC errors.

We further narrow the set of faults in those layers to those which we seek to automatically diagnose. As mentioned above, there are a wide variety of faults that we could have chosen to study, and our chapter on related work provides a broader discussion of the various classes of faults in Wi-Fi networks and specific examples of faults in each class. We chose to select a subset of these faults to diagnose based on our first-hand experience with the problems that commonly occur in the wild.

The four faults that we seek to diagnose are as follows:

1. **Coverage gap.** A device is located too far from any AP to achieve adequate performance.

2. **Sticky client.** A device fails to connect to an AP offering a stronger signal in favor of remaining associated with (or "sticking to") its current AP.

3. **Downlink oversubscription.** The downlink throughput required across all users of an AP exceeds the maximum throughput available on the channel.

4. **Uplink congestion.** The number of users attempting to transmit on the uplink to an AP causes excessive contention and reduction in realized throughput.

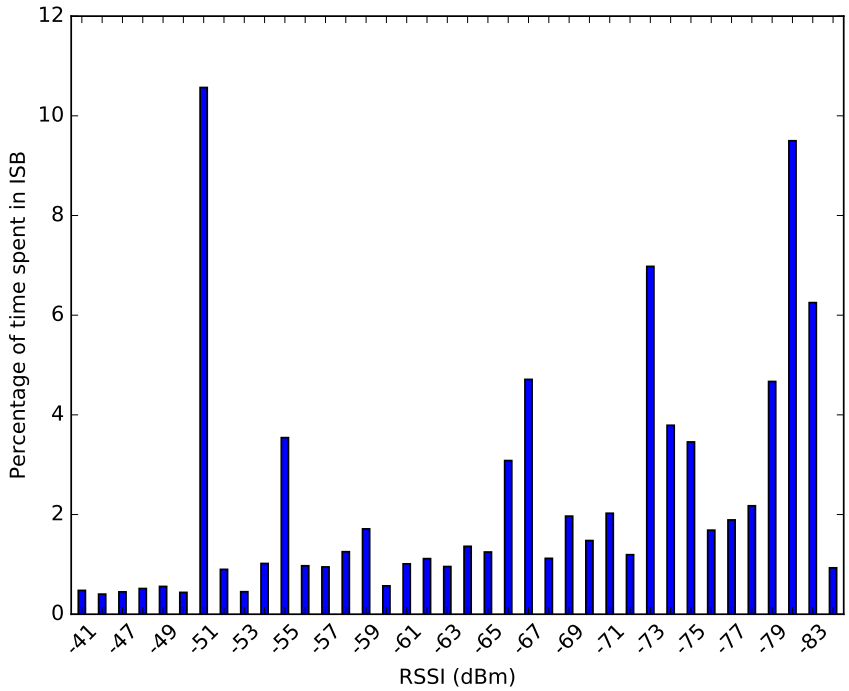

Figure 6.3: The duration ISB events in proportion to the total amount of video-playback time spent at each client-measured RSSI level. Similar to Figure 6.2, the slight upward trend toward the right side of the chart indicates that lower RSSI measurements correlate with more ISB errors.

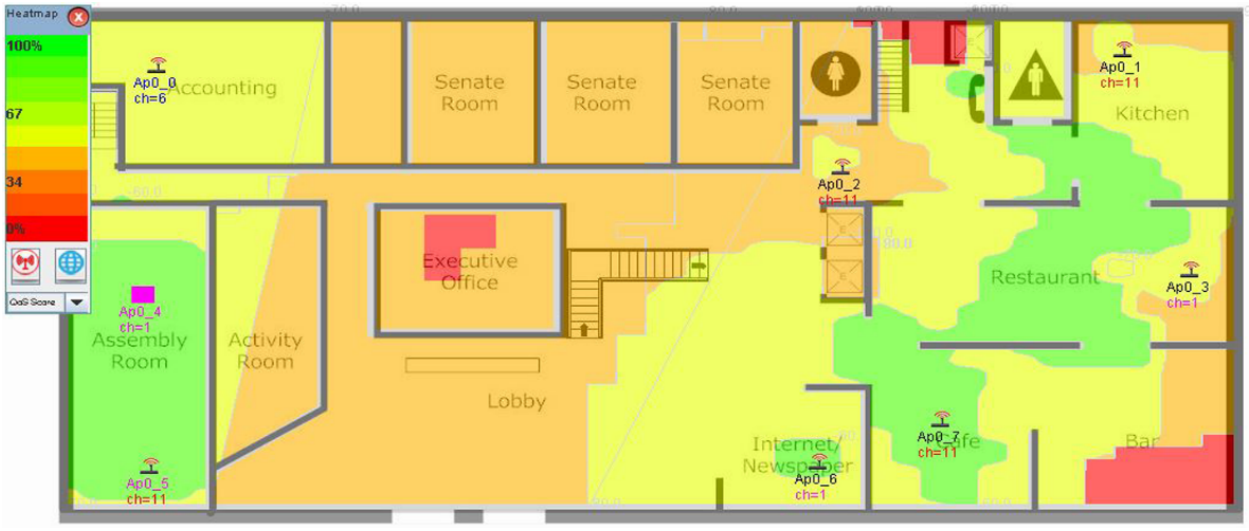

Figure 6.4: A visualization of Wi-Fi coverage in a building, showing areas with coverage gaps [76]. Areas are shaded darker corresponding with lower Wi-Fi signal strength. The darkest-shaded areas may be locations where a coverage-gap fault has occurred.

6.3 Fault Descriptions and Performance Impact of Faults

Coverage gap. The coverage-gap fault is perhaps the easiest to describe of all of the faults in our fault model. In a coverage-gap fault, a portion of the physical space covered by the Wi-Fi network is not receiving enough signal power from an AP to provide service. Figure 6.4 demonstrates this in a sample office environment. The areas shaded red (for example, a portion of the Executive Office) are locations where the received signal power of any of the APs in the office is zero. Coverage gaps are caused by access points not placed densely enough to cover a large space, or in some cases the building having areas where signals cannot reach due to physical obstructions. An example of the latter is in the lower-right corner of 6.4, where the signal from Ap0_7 is attenuated through the wall to the right of the AP and thus is unable to extend far enough into the adjacent room to provide coverage in the far corner.

The coverage-gap fault can manifest in other ways, as well. If the device remains connected while at the extreme range of the AP, that is also considered a gap in coverage (for example, the orange-shaded areas in 6.4). Devices in this situation may be forcibly disassociated from the AP due to minimum-link-speed thresholds set on the Wi-Fi network. These are often set to prevent devices from remaining associated to a distant AP when a closer one is available (sticky clients), as well as to prevent these distant devices from taking up an excessive amount of airtime (due to their slower link speed), thus affecting nearby clients as well.

To show the impact of coverage-gap faults, we studied the degradation of performance that devices suffer as they move away from their connected AP in a high-density Wi-Fi environment. As shown in 8.2, at a distance of 5m from the AP, all 20 devices exhibit good performance in the chunk-download simulation. Also, at 5m, the RSSI values measured by those devices ranged from -29 dBm to -44 dBm, which are very high values for a Wi-Fi connection. Note that, most large-scale Wi-Fi deployments, designers target a signal power of between -60 and -70 dBm from the nearest AP to the device.

Figure 6.5 shows the performance of devices as they are moved to the furthest position from the AP at 20 meters. At this point, the RSSI values measured by the 20 devices ranged between -55 and -70 dBm (with one device briefly measuring -80 dBm but quickly returning to -63 dBm). This is well within the acceptable signal-strength ranges targeted by large-scale Wi-Fi networks, and 20m from the nearest AP is not an unusual distance. However the signal power at this range was insufficient to allow any of the 20 devices to stay above the error-free threshold as shown in the figure. Clearly, switching to a closer AP is required in this case in order to maintain acceptable performance.

Sticky client. The sticky-client fault is located within the Wi-Fi driver itself. In a sticky-client scenario, the Wi-Fi driver remains associated with its current AP despite a closer AP being present and offering higher

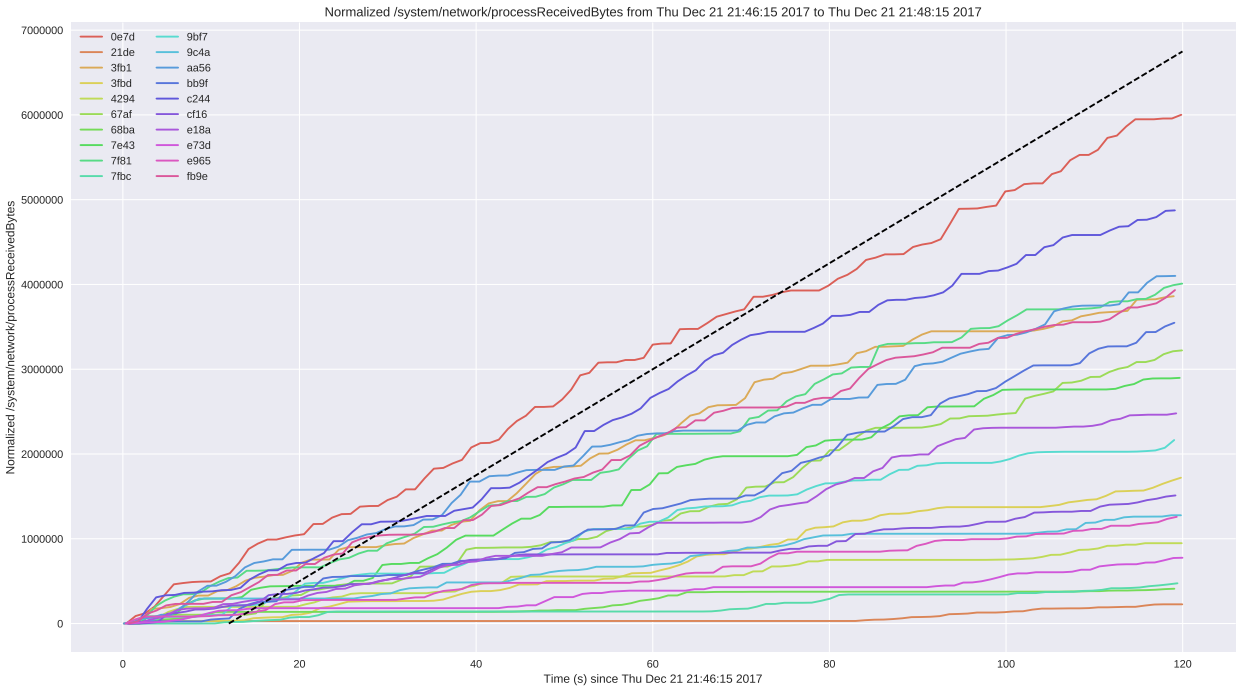

Figure 6.5: A graph showing chunk-download performance in our testbed with 20 devices positioned 20m from the AP, creating a coverage-gap fault. Each solid line represents the number of bytes downloaded from the network over time for a single device. In order to maintain error-free performance, the solid lines must stay above the dotted line. None of the 20 devices under test received enough throughput to remain above the error-free threshold throughout the 2-minute duration of the test.

signal strength. This could be due to the Wi-Fi driver not scanning for nearby APs or the driver being programmed to prefer to remain connected to its current AP.

We conducted experiments to show the prevalence and impact of sticky-client faults. Specifically, we sought to determine how well the driver can maintain Wi-Fi-performance to power our simulation while the devices are moved through each of the test positions and back, over a 2-minute period. To set up this test, we connected all of the devices to a SSID shared across 3 APs within an office building. We mounted 20 devices on a mobile cart and placed the cart approximately 2 meters away from one of the APs. We then ran a synthetic video-streaming workload on all 20 devices. Over the course of two minutes, we moved the cart past the locations of the other two APs, approximately 20 meters away from the first AP, and then back to the original position. We moved the devices at a speed of approximately one-third of a meter per second. Note that this is well below the typically preferred human walking speed of 1.4 meters per second, and so gives the device ample time to make roaming decisions.

Our results are shown in Figure 6.6. The devices exhibit error-free performance for nearly the first 60 seconds of the test, as they approach the 20m mark, at this point, 18 out of the 20 devices no longer receive enough throughput to keep their buffers full, and the buffers on 15 of the 20 (75%) eventually underflow,

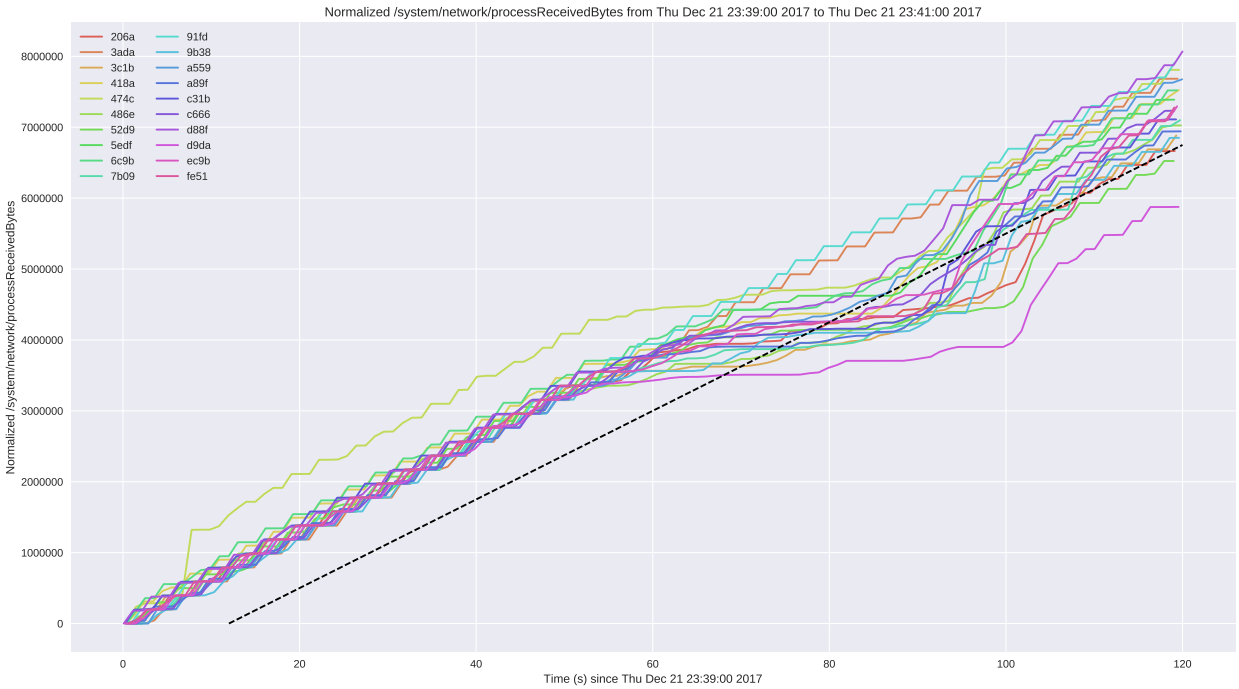

Figure 6.6: A graph showing chunk-download performance as 20 devices move throughout the 20m length of the test area using the driver's default AP-selection method. Each solid line represents the number of bytes downloaded from the network over time for a single device. In order to maintain error-free performance, the solid lines must stay above the dotted line. All devices initially had error-free performance, but as the devices moved to the 20m distance from the AP (a sticky-client condition), most devices were unable to maintain error-free performance.

which would interrupt the video-playback experience for the user. Fortunately, once those devices move closer to the AP, the buffers on all but two devices recover and resume error-free performance for the remainder of the run.

The 75% failure rate of devices is significant and shows the how the sticky-client fault can have an impact on the user's experience. It's true that a larger buffer may have masked some or all of the errors, but this buffer configuration already introduces 12 seconds of latency, and many video streams aim to achieve lower latency than that (especially at sporting events). Also note that there were other APs in the area (especially at the 20m position) that the devices could have used to recover video streaming, but most of the devices chose to remain connected to their original AP.

Downlink oversubscription and uplink congestion. The downlink-oversubscription fault is the result of resource exhaustion, where the throughput requirements across all clients connected to an AP exceeds the total airtime available on the channel. Under standard 802.11 MAC behavior it is possible, without any form of rate limiting, for a few devices to exceed the total bandwidth provided on a channel (usually in the hundreds of Mbps). This could occur if these devices are all using a high-bandwidth application, such

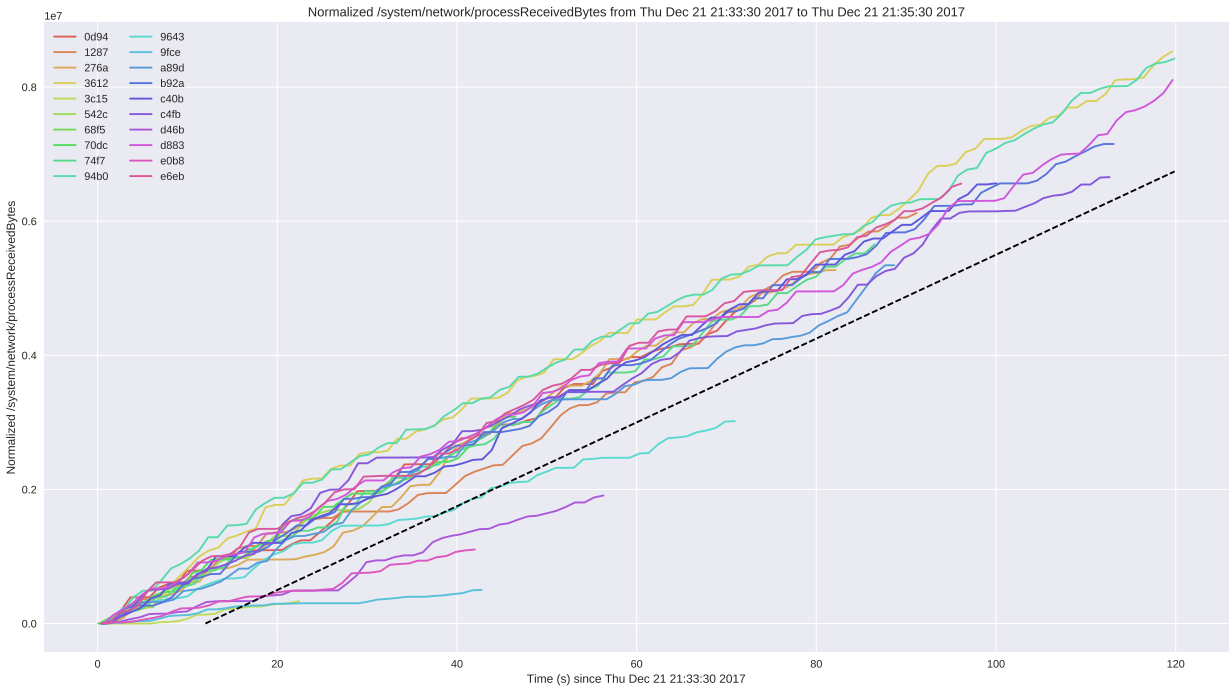

Figure 6.7: A graph showing poor chunk-download performance caused by a download-oversubscription fault. 20 devices positioned 5m from the AP were configured to download 750 KB chunks at 4-second intervals. Each solid line represents the number of bytes downloaded from the network over time for a single device. In order to maintain error-free performance, the solid lines must stay above the dotted line. Most of the devices maintained error-free performance initially, but many stopped receiving data before the experiment concluded (indicated by the end of the line occurring prior to the right edge of the graph).

as playback of high-definition video. In this case, the devices would be unable to obtain the bandwidth they require, causing an impact to performance.

The uplink-congestion fault is related to 802.11 MAC behavior and specifically the collision-avoidance mechanism [26]. When devices in a Wi-Fi network wish to transmit, they first briefly sense the medium, and if they sense no other stations transmitting, wait a brief period (the backoff period) and begin their transmission. As the number of transmitters grows, there is a greater probability that two devices will select the same backoff period and transmit at the same time as a result. This results in a collision (co-channel interference), where the transmitted frames from both devices are lost and the airtime used to transmit those frames is wasted, thus increasing network overhead. If this happens frequently (due to a large number of devices or frequent uploads), the network will suffer a form of congestion collapse, where the realized throughput is much less than the actual throughput otherwise available on the channel.

As with coverage gaps, we ran experiments to determine the impact of this type of fault on video-streaming performance. Figure 6.7 shows a test where chunk-download performance was poor, and several devices failed to download the entire stream during the 120-second window. Because the devices

were only 5m from the AP and no other traffic was using the network, we can conclude that the higher bit-rate of the stream saturated the channel capacity in the area. Also, the airtime utilization measured by the AP during this test approached 92%, which is the highest measurement we saw during our tests.

6.4 Key Indicators for Faults

We studied the above faults by injecting them in the laboratory on our Wi-Fi testbed. Details of this testbed, its construction, and our fault-injection strategy can be found later in the chapter on experimentation. This section focuses on the patterns in our data that indicate each of the faults in our fault model, and inform the development of our problem-diagnosis algorithm.

There are two primary defining characteristics of coverage-gap faults. The first is the lack of any APs for the Wi-Fi network within the scan range of the device. If no such APs appear, it is a strong indication that the device is located within a coverage gap. The second defining characteristic of a coverage gap is when all nearby APs have a low RSSI measurement, an a connection is attempted. This also indicates a coverage-gap fault, as it suggests that the device is located on the very edge of Wi-Fi network coverage.

Sticky clients are in a similar situation to devices within a coverage gap, being on the edge of Wi-Fi network coverage of the AP to which they are currently connected. However, the defining difference between coverage-gap and sticky-client faults are the presence of other APs nearby with significantly higher RSSI measurements. This means that a closer AP could be selected but the device is choosing to remain connected to the more distant AP, indicating a sticky-client fault.

Downlink oversubscription and uplink congestion are both related to the bandwidth consumed across all devices connected to the same access point. As described above, if the required download throughput across all devices exceeds the airtime available to the access point, a downlink-oversubscription fault has occurred. If instead there is not enough airtime available to service all of the clients' upload needs, this is a uplink-congestion fault. To simplify this, if there are several devices downloading data at a high bit-rate, this indicates a downlink-oversubscription fault. If there are several peers uploading data at a high bit-rate (combined), that indicates an uplink congestion fault.

6.5 Rule-Based Diagnosis

Our analysis of our fault model and the key indicators of our faults allowed us to define a decision tree that captures the process of diagnosing each of our faults based on the metrics available through our instrumentation system. We developed this tree by examining the impact that each of the faults in our

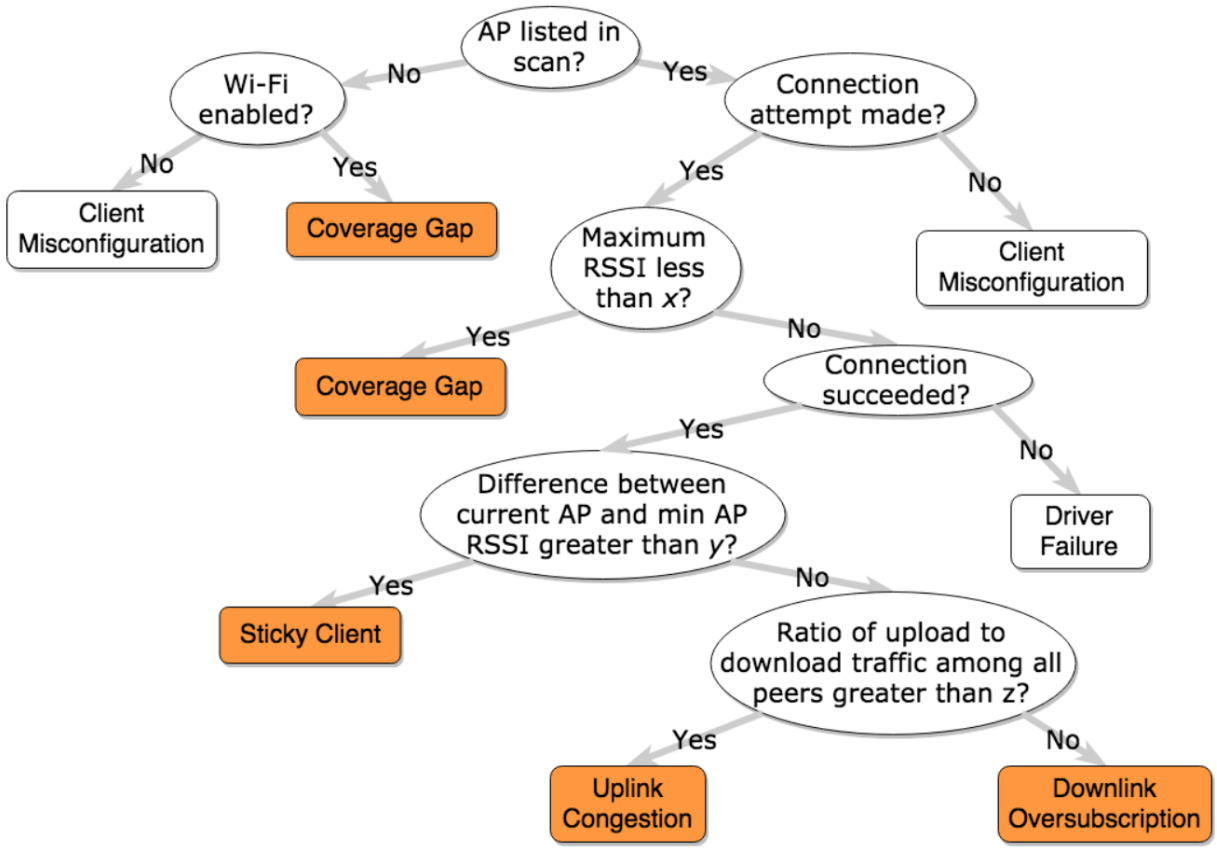

Figure 6.8: Our diagnosis decision-tree, based on our analysis of the features which indicate each of the faults in our fault model. This decision tree is evaluated by the end-user device when a problem is detected, in order to diagnose the most likely root cause of the problem.

fault model have on our instrumentation metrics, and then identifying decision points in our tree based on those primary discriminating factors. The resulting decision tree is shown in Figure 6.8.

Assumptions. Before describing the tree in detail, we wish to outline the assumptions that form the basis of our approach. We make a number of simplifying assumptions to aid in the construction of our decision tree:

- We assume that all devices are executing the same workload, which in our tests was our synthetic video-streaming workload. This allows us to base our decisions on the knowledge that each client is actively contributing to the load on the Wi-Fi network, or is attempting to, and is doing so roughly equally (i.e. no one client is attempting to hog most of the bandwidth).
- We assume that there are multiple devices near the device experiencing problems from which to gather data. This allows us to gather data from multiple perspectives in the network and use this data when diagnosing problems.

- We assume that the data we are receiving from all devices (including the one experiencing the problem) is accurate and there is no misinformation or malicious devices.

Description of decision nodes. The construction of our decision tree, specifically the ordering of the decisions in the tree, is influenced by the process that a device must execute in order to connect to the Wi-Fi network and obtain a video stream. To even attempt to join the Wi-Fi network, the device must receive a beacon from at least one AP serving that network. If that occurs, that AP will be included in the list of scanned APs. If it does not, it would indicate that either the device is out of range of any APs, or the Wi-Fi driver on the device is disabled (and thus is unable to receive any beacon messages). A simple test for whether the driver is disabled discriminates between those two cases.

Next, the device must attempt to make a connection (associate) with one of the APs in range for the Wi-Fi network. This involves contacting the AP and performing a two-way handshake defined in the 802.11 specification. If the device begins this process (regardless of whether the attempt is successful), the device will record a state transition to an "Associating" state. If the device fails to even do this, it indicates that the device is not configured to connect to the Wi-Fi network, thus indicating a client misconfiguration fault. Most Wi-Fi connected devices, including the ones we study, do not automatically attempt to connect to any open Wi-Fi network in range, and they must explicitly be told to connect to a network, either by an application running on the device or manually by the user. One exception to this rule is cellular-operator-managed Wi-Fi networks, to which devices locked to that carrier are typically pre-programmed to connect.

Assuming a connection attempt has been made, the remainder of the tree focuses on diagnosing problems with the communication between the device and the selected access point. Regardless of whether the device successfully connects, if the RSSI among all visible APs is too low, that indicates that the device is on the edge of the Wi-Fi network, thus indicating a coverage-gap fault. If that is not the case, we check whether the device has successfully connected to the network. If that is not the case, we diagnose a driver failure. If it has connected, we look for indications of one of our remaining three faults.

We next look at the RSSI measurements for not just our currently-connected AP but other APs in the area as well. If the currently-connected AP has a much lower RSSI measurement than another AP in the area, that indicates that the Wi-Fi driver is choosing to remain connected to an AP with a weaker signal when an AP with a much stronger signal is available. We then diagnose a sticky-client fault.

If the device is connected to an AP with strong signal, we then look for our final two faults which relate to the utilization of the Wi-Fi channel itself. At this point, we know the most likely fault is one that impacts the throughput of the Wi-Fi channel, since the device has successfully connected with a nearby

AP that is one of the strongest in range. To diagnose the problem further from this point, we need more information about other nearby devices. Once we have gathered this information (the process of which is detailed later), we look at the number of devices connected to the same AP to which we are connected. Recall our assumption about all devices running identical workloads. If the workload is skewed toward download bandwidth (meaning there is more download traffic than upload), the most likely root cause is downlink oversubscription. If instead there is more upload traffic than download, the most likely cause is uplink congestion.

Implementation. We implemented our decision tree by programming the decision-making logic directly into our diagnosis system. The algorithm accepts as input a periodic sliding-window summary of our instrumentation framework's measurements just prior to the detection of a problem. We leave the size of this window as a variable to be determined later. This gives the algorithm both the latest data along with a summary of the historical data just prior to the time that the problem occurred on which to base its decisions. Some of the decision points in the tree are simple questions such as "Is an AP in the scan list?" and "Is Wi-Fi enabled?" These questions can be answered by looking at the most recent state summary. Some of the questions require a review of the historical state, such as determining maximum RSSI values across all APs and the difference between RSSI measurements. We look at the average of these values over some time prior to the problem, as a user-visible problem may not manifest until some time after the fault occurred due to buffering of application-layer data.

Runtime operation and proximity detection. Our diagnosis engine is triggered when our problem detector, which is continually monitoring video playback on the device for problems, identifies a problem during playback of the video stream. At this point, it begins acquiring data from as many sources as possible to provide multiple perspectives of the network for our diagnosis algorithm. Once it has gathered enough data, it executes the decision-tree algorithm described above to decide the most-likely fault among the ones in our fault model. This is then output as the result of the diagnosis, and a mitigation strategy is applied (if any). At that point, the problem detection resumes, and if problems persist the problem-diagnosis engine runs again, and the process repeats.

The data sources used during the execution of the problem-diagnosis engine have changed significantly over the course of the development of our approach. As we developed our approach, we elevated the importance of some sources of data and reduced or eliminated other sources entirely. We initially used data from two perspectives of the network, the client impacted by the problem (the client-side perspective) and the access-point to which that device was associated, if any (the infrastructure-side perspective). While this does provide the necessary data to execute our decision tree, we found that, due to the latency in the data that we can retrieve from the access point (approximately 5 minutes for many metrics), the

infrastructure-side data was not granular enough to be able to associate readings of infrastructure metrics with the occurrence of a problem.

To work around this problem while still taking advantage of data from multiple perspectives, we introduced the notion of proximity into our device tracking. By proximity, we mean understanding which devices are near a device by physical distance. One method of proximity determination involves defining a function which maps a device to one of several discrete locations (usually regions of a building). All of the devices within that region are considered to be in close proximity to each other. Using the data we have available, the simplest method of defining this function is to simply map each device to its currently-connected AP. Since APs are spread throughout the building, each AP roughly defines a circular location centered on the AP with radius proportional to the signal power of the AP. However, this definition has two drawbacks: first, AP regions can overlap, and second, signal propagation from an AP is often irregular due to the positioning of walls and objects in the physical space.

A better method of defining proximity is to manually map out regions of the physical space where devices could be located, and then at runtime attempt to determine which regions the device is located within using Wi-Fi fingerprinting [28]. Wi-Fi fingerprinting is a technology typically used for indoor positioning which uses a RSSI measurements to multiple nearby APs to approximate the device's location. Such system use databases of location information learned from site surveys, where system administrators walk through the building with devices to gather Wi-Fi signal-strength measurements at various locations. To determine the approximate location of an unknown fingerprint, the system compares fingerprint against the fingerprints in the database with known locations, usually using some distance metric on the RSSI values. The location of the fingerprint with the smallest distance is selected as the location for the unknown fingerprint.

This method is a significant improvement over just using the currently-connected AP, but has its own drawbacks. First, it requires the introduction of a database of location measurements, and a team to build this database. This can be an expensive and time-consuming process, especially for a large building. Also, the location database must be online and accessible to the diagnosis algorithm at all times in order to make location determinations. If the diagnosis algorithm is running on the client because it is unable to connect to the server, it will be unable to determine its location or download data of other devices nearby, thus not being able to take advantage of multiple perspectives.

The final evolution of our approach is to eliminate the reliance on infrastructure to provide other perspectives of the Wi-Fi network. The primary motivator for this approach is the fact that a device experiencing a problem with the Wi-Fi network is unlikely to be able to send or receive diagnostic information over that network. Instead of relying on Wi-Fi fingerprinting (and thus a possibly-unreliable Wi-Fi

network) to determine proximity, the device attempts to contact nearby devices directly using the GATT function of Bluetooth Low Energy (BLE) communication [33]. At startup, the device uses the BLE subsystem of Android (now commonly available on all Android smartphones [35]) to advertise its presence to nearby devices, publishing a GATT service indicating our wireless diagnostics are available within that advertisement. When our problem-diagnosis engine runs, the device again engages the BLE subsystem on the device to scan for advertisements. For each advertisement seen, the device attempts to connect to that device and download its sliding-window state summary (containing a summary of all of the instrumentation metrics tracked on that device). Included within this summary is the amount of data uploaded and download from each access point, which is used as part of our diagnostic algorithm. This peer-to-peer approach thus provides us with data from multiple perspectives of the network, all without relying on any Wi-Fi infrastructure to provide the data.

6.6 Threshold Determination

In the description of our diagnosis algorithm in the previous section, we left many of the thresholds undefined. This is because those thresholds are unique to each Wi-Fi deployment, and setting them to specific values would provide an approach that is too brittle to be used generally across multiple networks without reconfiguring those parameters. In this section, we outline an approach to determining those thresholds for a given Wi-Fi network.

Mapping a continuous or integer-valued range of inputs to a simple yes/no question suitable for a decision-tree node is a process known as discretization [25]. To learn the thresholds for a new Wi-Fi network (also known as splits in discretization), we must perform fault injection on network in question in order to collect a set of labeled training data. This training data provides measurements of the network when each fault is introduced, allowing us to determine the split that optimizes the accuracy of our classification.

For each node in the tree requiring discretization, we use the binary-discretization method described by Fayyad and Irani [29]. In this algorithm, to determine the best value for a threshold, the training examples are first sorted by the values of the attribute in question. Then, for each possible binary partitioning of the data, the value of the attribute on either side of the split point is set as the threshold in the decision tree, and the output of the tree is evaluated for each example. To measure the accuracy of the split, the number of correct classifications in the training set out of the total number of classifications performed is calculated. After calculating the accuracy for each split, the split with the highest accuracy is chosen as the value for the threshold, and the process repeats until all thresholds are determined.

Chapter 7

Problem Mitigation

In this chapter, we present our problem-mitigation approach. The primary method of problem mitigation, when it can be performed in an automated manner, is to associate the device with a different access point. This comes with a set of challenges, in both selecting which alternate access point to choose (if any are available), and implementing a method to force a device to select a different access point.

7.1 Driver-Based Access-Point Selection

We begin with a brief discussion of driver-based access-point selection. All major operating systems and drivers today use essentially the same access-point-selection algorithm: Strongest Signal Strength (SSS) [56]. This algorithm effectively scans the network for some period of time, and at the end of the scan period chooses the access point with the highest signal strength. A study of SSS behavior by Nicholson et. al. [56] showed that the performance of the SSS approach was no better than choosing access points at random.

Furthermore, once an access point is selected, Wi-Fi drivers are often reluctant to switch between access points even if a better AP is available [68], known as the sticky-client problem. This problem presents a challenge to high-bandwidth/low-latency services, such as video streaming that are used while a user is mobile on a Wi-Fi network. As discussed in the presentation of our fault model, we performed an experiment to replicate sticky-client behavior on our testbed, and found that this behavior caused user-visible video-playback errors on 75% of our test devices (see Figure 6.6).

There have been other AP-selection mechanisms proposed in academic literature as recently as 2017, and some have been included in the 802.11 specification. The 802.11k extension to the 802.11 specification [41] provides devices with more information on nearby access points, such as the amount of load or channel utilization at that AP, so that they can use in conjunction with RSSI measurements to make better

AP-selection decisions. Furthermore, recent studies [47] propose better client-side, driver-based selection algorithms that do not require more information from the access point.

7.2 Overriding Driver Roaming-Decisions

In order to effectively mitigate problems related to a device's current AP association, we need a way to force the driver to choose a different access point and remain connected to it. We attempted to find a reliable mechanism to do this on the Android platform, and we discuss our experiences with doing so in this section. While Android does provide such a mechanism, we were unable to use it to reliably achieve our goals, and for the purposes of our research opted for a different approach that requires cooperation from the Wi-Fi network.

Wi-Fi networks in Android can be configured programmatically using the `WifiManager` interface [34]. Within that interface, the `addNetwork` method accepts a `WifiConfiguration` that allows the programmer to define exactly which network the device should connect to. The parameters in this object include the SSID, security options (e.g. the WPA2 passphrase), and an optional BSSID. The documentation for the BSSID parameter states "When set, this network configuration entry should only be used when associating with the AP having the specified BSSID." Since the BSSID uniquely defines an access point (or, to be precise, a radio on that access point), it is expected that setting this parameter and activating this Wi-Fi configuration would force the device to connect to the indicated BSSID, and only that BSSID.

However, this is unfortunately not the case. In practice, setting that parameter and activating that Wi-Fi configuration causes Android to exhibit a number of buggy behaviors. Among these, is that activating a configuration with the BSSID specified causes the operating system to display two configurations for the network, one with the BSSID set and one without, and it becomes impossible to manually delete both configurations through the settings UI. The only way to resolve this situation is to reboot the device, which clears the second configuration. Additionally, even if the configuration with the BSSID set is active, the Wi-Fi driver will often switch away from the selected BSSID a few seconds later, as it makes its own AP-selection decisions independent of Android (on the devices we used to test).

We investigated the source of this problem, and found that the mechanism to force roaming (or rather, the ability to disable the driver's internal AP-selection mechanism) is not available on the Wi-Fi drivers installed on our devices. It is exposed as a capability for those Wi-Fi drivers that do support it (`WIFI_FEATURE_CONTROL_ROAMING`), but this was not available on our test devices. Unfortunately, we were thus unable to control roaming using BSSIDs alone.

To work around this problem for the purposes of our experiments, we set up multiple SSIDs within

the testbeds we used when testing fault-injection or problem-mitigation scenarios that require fine-grained control over AP selection. Essentially, each AP was assigned a unique SSID, and on startup of our program, the program forced the device to connect to a particular AP by activating the Wi-Fi configuration for that SSID. When we needed the device to switch to a different AP (BSSID), we changed the SSID instead. Note that, while in this mode, there is no increase in management-traffic overhead due to the use of multiple SSIDs, as each AP is only sending beacons for a single SSID. In a standard configuration, each AP would still be sending beacons for just one SSID, except it would be the same across all three APs.

7.3 Access-Point Selection with Fine-Grained Location Tracking

One approach we explored to improving access-point selection uses Wi-Fi fingerprinting [28] to override Wi-Fi-driver roaming decisions. Wi-Fi fingerprinting is a technology that uses RSSI measurements to nearby APs to estimate position with a precision higher than could be obtained by GPS. Using the location obtained from Wi-Fi fingerprinting, when executing problem mitigation, we check that the currently-associated AP is the one covering the device's current location. If this is not true, we force the device to switch to that AP.

To implement this approach, we send the latest set of scan results (the Wi-Fi fingerprint) from the device to a web service that executes Wi-Fi fingerprinting on those scan results and returns the SSID corresponding to the device's location. Internally, the service uses a naive Bayes classifier. Prior to use of the system, this classifier needs to be initialized by moving a device through each location for approximately 5 minutes to present training data to the classifier. Once trained, the classifier provides high-accuracy location data when presented with a Wi-Fi fingerprint. Each physical location is tagged with the SSID and BSSID of the AP for that location, so the location returned by this service also provides the SSID/BSSID information of the AP servicing this location.

The service has three primary functions related to detecting and mitigating Wi-Fi problems. The first is determining location using Wi-Fi fingerprinting, allowing the service to determine the nearest AP to the device's current location. The second is ranking each nearby AP according to an estimate of the throughput it can provide (abstracted as a numeric "score" assigned to each AP). The third is forcing the device to switch to a better AP if another AP has a higher ranking (by a significantly higher score margin) than the currently-connected AP. The former is accomplished by presenting the service with the results of the latest Wi-Fi scan. When calculating each AP's score, the AP for the current location receives a flat score bonus, and then an additional value is added to or subtracted from each AP's score based on the RSSI to that AP. If after a scan is scored, the score for a nearby AP is significantly higher than the score

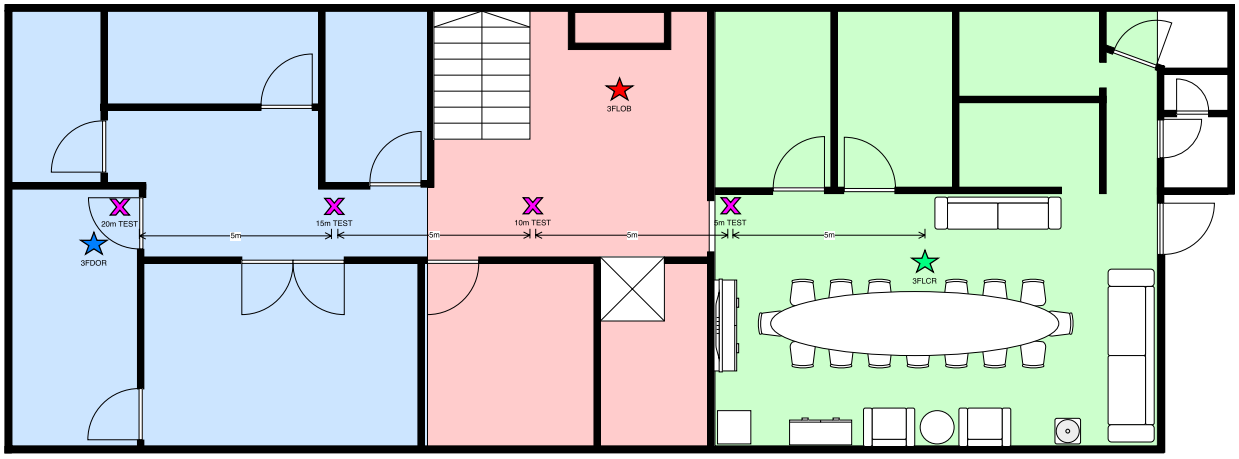

Figure 7.1: A diagram of our Wi-Fi testbed for problem mitigation via fine-grained location-tracking, showing AP positions, AP coverage regions, and test locations.

for the current AP (above a configurable threshold) and the device has not switched APs recently (also a configurable threshold), the service forces the device to switch to the more favorable AP by reconfiguring the Wi-Fi settings of the device as described above.

Testbed. Figure 7.1 shows a diagram of our testbed. We worked with a local Pittsburgh software-development company to use their offices and Wi-Fi infrastructure for our testbed, and the diagram shows the third floor of this office building. Three access points (model Unifi UAP-AC-PRO) service the floor, with each access point's intended coverage area covering approximately one third of the floor. These AP locations and coverage areas are shown on the diagram and labeled as 3FDOR (blue), 3FLOB (red), and 3FLCR (green). The labels correspond to the internal naming of the access points, and this naming was also used in our experiments. All three APs used a configured transmit power of 25 dBm. Channel assignments were configured automatically by the APs based on the local RF environment, resulting in 3FDOR and 3FLOB using channel 1 while 3FLCR used channel 6. We also defined four testing locations at fixed distances from the 3FLCR AP at 5, 10, 15, and 20 meters.

Each AP was configured with two SSIDs to support our testing. The first was a SSID common across all access points, used to test the default driver-roaming behavior. The second was a SSID unique to that AP, used to test forced roaming between APs (from our code). Because the office's enterprise Wi-Fi network used the 5 GHz band, we were unable to add these SSIDs to that band. Therefore, all of our tests used the 2.4 GHz band exclusively.

In addition to the Wi-Fi infrastructure, the testbed included 20 Android Nexus 7 tablets (test devices) running OS version 5.1.1. Installed on each of these tablets was a mobile app that included our instrumentation code, our problem-mitigation engine, and a simulated video-streaming application. Through the

testbed's Wi-Fi infrastructure, we implemented an control system that allows us to perform several tasks on the test devices simultaneously, including installing and uninstalling apps, launching and terminating apps, and clearing data stored locally on the device. We used this control system to set up, start, and stop our experiments. We mounted all 20 devices on a cart that allowed us to easily move all of these devices between test locations.

Evaluation. We tested the performance of this approach in our testbed. We configured up our 20 devices 2m away from the starting AP and moved those devices back and forth over 20 meters at a speed of about 0.5 meters per second, through three location zones serviced by two other APs. During the experiment, the devices were running a video-playback simulation to generate a synthetic workload. As the devices moved throughout the experiment, they reported their Wi-Fi fingerprint periodically to the location server, and the server responded with the AP with the highest score. When the server's response indicated that another AP had a higher score than the current AP, the device initiated a forced switch to the SSID corresponding to its location. To prevent rapid bouncing between APs when on the edge of a region, the device only initiated this forced switch if it had not done so previously within the last 15 seconds.

Our results are shown in figure 7.2. Here, 7 of the 15 devices experienced failures downloading chunks upon their first AP switch about 30 seconds in to the test (corresponding to movement of about 10 meters). These devices did not recover for the remainder of the test. A further 6 devices experienced failures during forced roaming that caused buffer underflows, but eventually recovered and continued downloading chunks (although a portion of the stream would have been missed). The remaining 7 devices managed to stay above the error line for the duration of the test. Overall, errors were visible on 13 of 20 devices, or 65%.

The 65% error rate is lower than the 75% error rate under default Wi-Fi driver behavior and does indicate that, overall, the forced-roaming approach performed slightly better than the driver. Also, the devices began to switch to closer APs much faster than in the driver-controlled case. However, forced-roaming caused a number of devices to fail completely without recovery (the recovery rate for the driver was 90%, versus 65% for forced roaming). Furthermore, for those devices that recovered after failure, the recovery time was much longer than in the driver-controlled case.

Clearly, forced-roaming based on location data has the potential to improve on the performance of the default Wi-Fi driver's AP-selection strategy. As suggested by our real-world data and anecdotal evidence, the Wi-Fi driver tends to exhibit stickiness as long as RSSI values remain above an acceptable level, but this behavior may not provide acceptable performance in a large-scale Wi-Fi environment. Location-assisted roaming allows devices to switch sooner to APs that likely can provide better performance, and our results show an improvement in the overall error rate using forced roaming over the default roaming

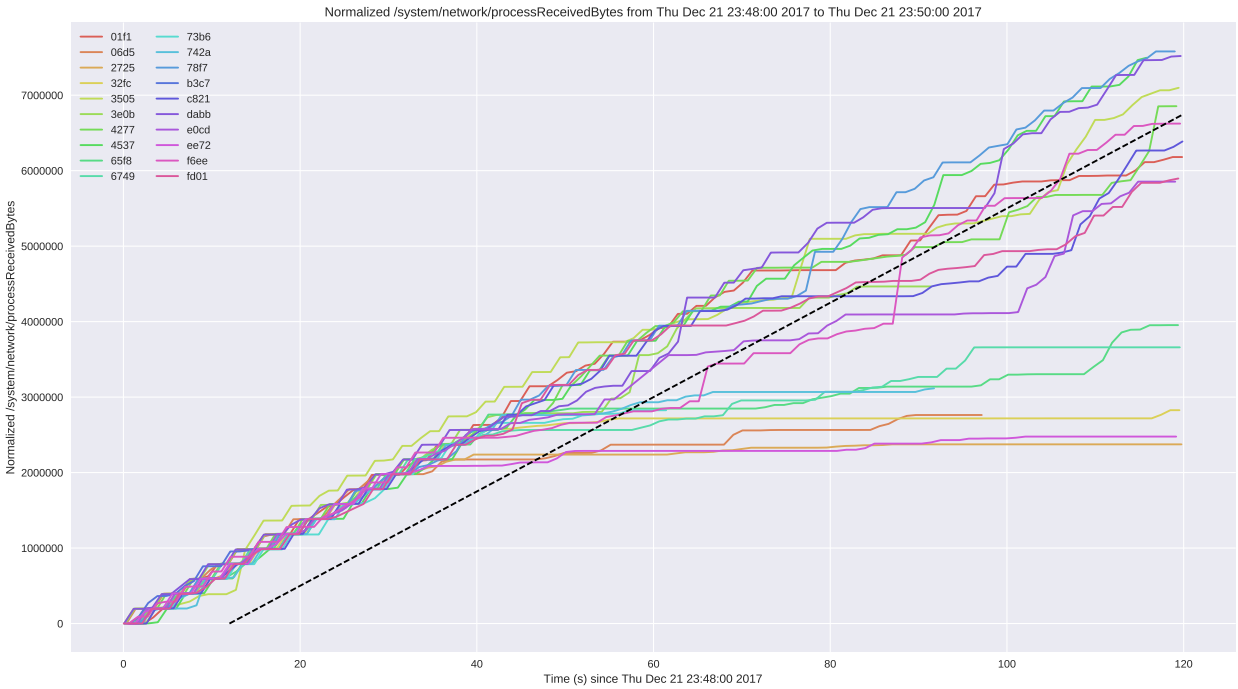

Figure 7.2: A graph showing chunk-download performance as 20 devices move throughout the 20m length of the test area using location-assisted AP-selection. Each solid line represents the number of bytes downloaded from the network over time for a single device. In order to maintain error-free performance, the solid lines must stay above the dotted line. Most of the devices maintained error-free performance initially, but when our problem-mitigation algorithm forced the devices to connect to a nearer access point, many of the devices stopped receiving data (indicated by the lines not reaching the right side of the graph).

behavior. However, our approach also exhibits undesirable side effects in terms of recovery rate and recovery latency. For this reason, we decided to implement a different approach to access-point selection that uses other perspectives of the network to make improved AP-selection decisions.

7.4 Performance-Aware Access-Point Selection

The final approach we studied takes advantage of other perspectives of the network from peers in order to make improved AP-selection decisions. Our problem-diagnosis approach has access to (and uses) data from several peers of the device experiencing a performance problem. This data effectively captures the historical and current state of the device across several Wi-Fi-related metrics. Among these are indicators of which APs to which the device was associated, along with a measurement of how long the device was associated to each AP. Furthermore, the measurements also include the bandwidth seen on each AP, over the window of the measurement, as well as the number of problems detected on that device while connected to each AP.

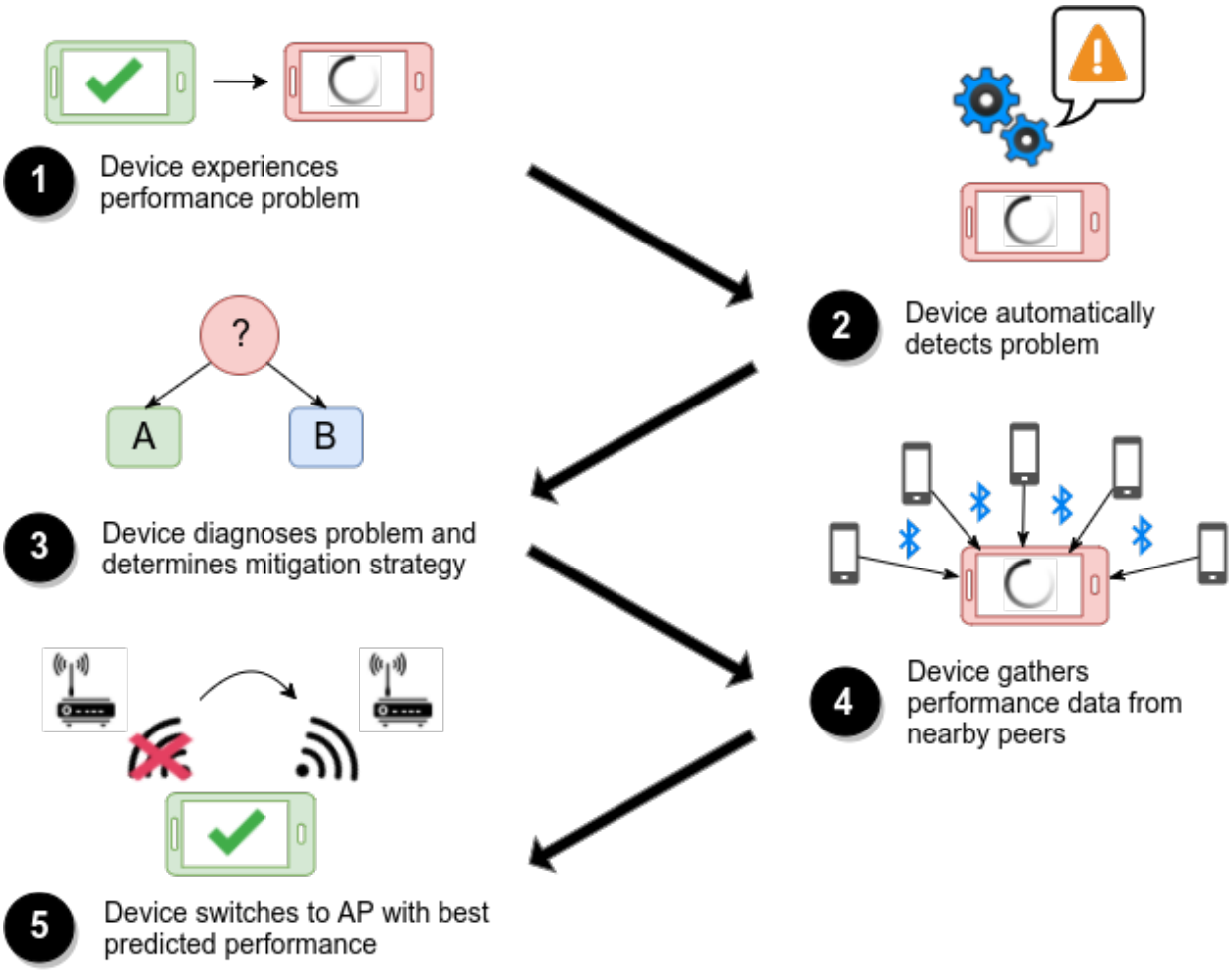

Figure 7.3: Operation of the performance-aware mitigation strategy.

The operation of our performance-aware problem-mitigation strategy is shown in Figure 7.3. The algorithm proceeds in three major stages. The first is problem detection, where the device is continually monitoring the performance of the application for user-visible problems. As discussed in previous chapters, for our instrumented video-streaming application, we consider buffering events longer than 6 seconds (in-stream buffering) or 12 seconds (initial buffering) to be user-visible problems. Once a problem has been detected, we diagnose the root-cause of the problem and determine if there is an automated-mitigation strategy that can be applied. If that mitigation strategy involves selecting another access point, we use a combination of highest-RSSI and performance-aware mitigation as appropriate, depending on the information available. Once an alternative access-point is selected, we force the device to associate with that AP, thus mitigating the problem.

In this section we focus on mitigating problems by switching to a different access point, which involves

a complex automated decision (best-access-point determination) and covers three of the most important faults in our model: sticky client, downlink oversubscription, and uplink congestion. The other faults in our model either have no automated mitigation or have trivial client-side mitigation strategies. An example of the former is the coverage gap, for which the only mitigation strategy is to physically move the device further within range of the Wi-Fi network's access points. Examples of the latter are client misconfigurations and driver failure, which require either correcting the device's configuration (e.g. turning on the Wi-Fi subsystem or setting a correct password) or, in the case of driver failure, resetting the Wi-Fi subsystem.

Once the device has detected a problem and the root cause has been identified as one of sticky client, downlink oversubscription, or uplink congestion, the device begins the process of determining the best alternative access-point with which to connect. It does this in two phases which will be discussed in detail below. In the first phase, the device connects to nearby peers using Bluetooth Low Energy (BLE) and requests the latest performance-data summaries that they have collected, based on their own experience with the Wi-Fi network. After collecting this data, the device determines if enough data is available from peers to make a performance-aware decision; if not, the device falls back to using the strongest-signal-strength approach. Once the device has chosen an alternative access point, it proceeds to force the operating system to disassociate from its current access point and associate with the newly-selected access-point.

The first phase of performance-aware access-point selection involves collecting performance data from nearby devices (peers). We studied multiple approaches to this problem. The first was to collect the performance data from all devices in a centralized location, which any other device can query for data from devices connected to nearby access points (with nearness determined by those access points listed in the device's recent scan results). This approach has two major drawbacks: first, devices must be able to connect to the central server in order to upload or download information, which may be difficult or impossible if the device's Wi-Fi connection is unreliable; second, Wi-Fi scan results, even if ordered by RSSI, are known to be a poor indicator of proximity [42] without additional supporting data and advanced algorithms.

Instead of a centralized, server-based approach, we opted for a decentralized approach to collecting performance data. As of this writing, nearly all smartphone hardware and operating systems provide support for Bluetooth Low Energy (BLE) [35], with the device able to act as both a central (client) or a peripheral (server), even simultaneously. We take advantage of this in our approach, by having each device equipped with our problem-diagnosis system advertise its availability for BLE communication to nearby devices. Since BLE is designed for short-range communication, we can assume that two devices are

in close proximity if they are within range of each other's BLE advertisements. Even when no problems are occurring, devices running our approach constantly track their peers by listening to these periodic advertisements.

In addition to sending out advertisements, devices are also continually tracking their own perspectives of the network and recording it in a concise form that is ready to provide to a peer as soon as it is requested. This is done by tracking a vector of performance metrics for each access point. Each element of the vector indicates the value of a performance metric for that vector's access point, with some values being averaged over a sliding time-window (by default 60 seconds). Each of the values are scaled so that each one can fit into a single byte (0-255), and currently the system tracks twenty metrics for each access point (meaning that the performance data for each access point requires twenty bytes of storage). It was necessary to keep the amount of per-AP performance data as small as possible so that the data for all access points will fit within the maximum size of a BLE-characteristic value, which we determined experimentally to be 360 bytes for the BLE implementation on our test devices. This allows each device to provide performance data for up to 18 nearby access points to its peers, which usually far exceeds the number of nearby visible access points. To keep this data fresh for peers that need it, the performance metrics for each AP are updated at 5-second intervals.

When a problem does occur, the device experiencing the problem initiates BLE communication with its peers in order to download their perspectives of the network. As required by the BLE GATT protocol, for each peer the device connects to that peer, discovers the services and characteristics for that peer, and reads the characteristic corresponding with the performance data that the device wishes to obtain. In our system, there is simply one service and one characteristic, and all devices are programmed to provide the entirety of their performance data when that characteristic is queried. As described above, this performance data contains 20 metrics from up to 18 nearby access points.

Among the 20 metrics are two that we use for performance-aware problem-mitigation: the number of buffer underruns (thus indicating the number of problems experienced by that peer) and the number of video chunks successfully downloaded (thus indicating the network performance for the video stream). To determine the best access point to use, we sort each access point in order first by the total number buffer underruns detected on that access point (ascending), then, to break ties, by the total number of video chunks successfully downloaded from that AP (descending), and as a final tiebreaker by the total number of bytes downloaded from that AP (descending). We then select the AP in the first sorted position. If the access points in the first and second sorted positions are equal in all metrics, one of the tied access-points is chosen at random.

There are cases where our performance-aware access-point selection can fail. The first is if there are

too few peers nearby, in which case the device doesn't have access to enough information about the performance of nearby access points in order to make a decision. The second is if there is not enough access-point diversity among the peers (for example, if all peers are connected to the same AP as the device experiencing the problem). In these cases, we fall back to the strongest-signal-strength mitigation-strategy, where the device simply selects the alternative access point with the highest signal strength. If no other access point are available, mitigation fails and the device remains connected to its current access point.

If an alternative access-point is available, the final phase of our mitigation approach is to force the device to connect to that alternative access point. As discussed elsewhere, the hardware and drivers on our test devices did not allow overriding access-point selection on a per-BSSID basis. For the purposes of our experimentation, we constructed our Wi-Fi testbed so that each access point broadcast a unique SSID, and to force association with an access point, our system instructed the operating system to connect to the SSID corresponding with that access point. We hope that future versions of the Android operating system and Wi-Fi drivers allow application software to control access-point selection on a more granular level so that our approach can be implemented across a wide variety of Wi-Fi networks.

Chapter 8

Experimental Evaluation

8.1 Testbed, Synthetic Workload, and Fault Injection

We evaluated our approach on a testbed designed to simulate the conditions found in a large-scale Wi-Fi network: multiple APs covering a large indoor area (with each AP designed to cover a specific section of the space), multiple clients located throughout the space, and external interference from other devices using the ISM bands (both 802.11 and non-802.11). Using an mobile app on Android devices, we simulated a high-bandwidth, low-latency video-streaming application running on this testbed, to mimic the type of demanding applications typically used on large-scale Wi-Fi networks. We then conducted several experiments to determine the performance of our diagnostic approach when faced with common Wi-Fi network faults, as well as the ability of our mitigation strategy to successfully resolve the problem.

Testbed. Figure 8.1 shows a diagram of the testbed we used to evaluate our problem-diagnosis system. We deployed our testbed on a single floor of an office building spanning approximately 30 meters in length. We placed two access points (TEST-2 and TEST-6) at each end of the floor, approximately 25 meters apart, to take advantage of as much signal attenuation between the access points as possible. We also placed a third access point (TEST-4) about 5 meters away from TEST-6 to facilitate testing of faults which do not rely on signal strength and also for our later experiments with problem mitigation.

The RF characteristics of our testbed are described in table 8.1. Our access points were capable of setting signal power at three different levels: high, medium, and low. These different settings produced different signal-strength (RSSI) measurements along the axis between TEST-2 and TEST-6. The Near position was measured approximately 1 meter from the access point, the Center position at the Center Point shown in Figure 8.1 (approximately 12.5 meters from both access points), and Far was measured approximately 1 meter from the opposite access point. This gives an approximation of the signal-strength gradient between TEST-2 and TEST-6, which we use to inject faults related to signal strength (coverage

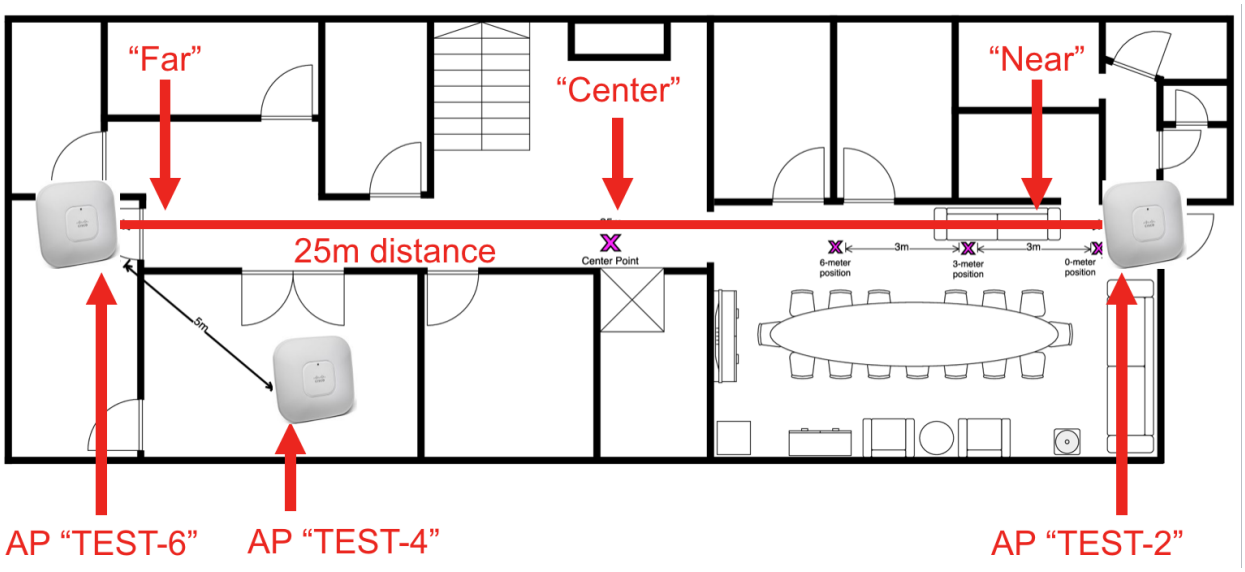

Figure 8.1: A diagram of our problem-diagnosis Wi-Fi testbed, showing AP positions and test locations.

gap and sticky client).

In addition to signal power, we configured each access point to broadcast a SSID. This SSID varied depending on the fault being injected, either with each access point broadasting a unique SSID according to its name (TEST-2, TEST-4, or TEST-6), or each access point broadcasting the same SSID (TEST-LB, for load-balancing). The SSID was open-access, not requiring any authentication to connect. We conducted our experiments during times when the office space was empty in order to avoid other devices being active within the space.

Each access point used a separate channel in the 5 GHz band. We chose channels 100, 104, and 108 with 20 MHz bandwidths. This ensured there was no overlap between channels. These channels are also not frequently used as they are dynamic frequency selection (DFS) channels, and we verified that indeed no other devices within range were using these channels. DFS is a mechanism which detects the operation of radar systems on the same channel used by Wi-Fi devices and causes the Wi-Fi devices to switch to a different channel if radar is detected. No radar systems were operating on these channels near our testbed's location, so the DFS function did not take effect.

Synthetic workload. The video-streaming application running on our testbed was designed to simulate a typical chunk-download video-streaming protocol. Common streaming protocols in this class include Apple's HTTP Live Streaming (HLS) and MPEG DASH. These protocols work by downloading live video in fixed-length chunks over HTTP or HTTPS. This can be extended in various ways, e.g. to support multiple bit-rates or different video codecs. While these protocols have been optimized in various ways to improve performance, we chose to assess the performance of our solution using a most basic implementation of

these types of protocols so that our approach remains applicable to all video-streaming protocols in this class and we do not optimize for a specific protocol or implementation.

Our simulation focuses on the core chunk-download portion of the protocol. The simulation is initialized with three parameters: S, the size of each chunk (in bytes), L, the length of each chunk (in seconds), and C, the capacity or maximum number of chunks in the device's video buffer. Attached to the testbed's LAN via a wired connection is a web server configured to provide files of size S (we used nginx 1.4.6 serving fixed-length files from disk). On startup, the video simulation attempts to fill its buffer by sequentially downloading C chunks of size S from the server. Every L seconds, the simulation removes one chunk from its buffer and attempts to download a new chunk from the server if a chunk download is not in progress. Once a chunk download completes, the buffer size is incremented by 1. if the buffer size is less than C, the simulation attempts to download another chunk immediately; otherwise, the simulation pauses and waits for the next chunk to be removed from the buffer before continuing. Based on our experience with typical low-latency video-streaming configurations, we fixed $L = 3$ and $C = 4$ for a total buffer capacity of 12 seconds of video.

Baselines and calibration. Before testing the performance of the stock Wi-Fi driver versus our approach, we tuned our video-streaming simulation for the RF environment in our testbed. As mentioned above, we performed our tests in a public office building on the 2.4 GHz band. Although we ran our experiments in the evening to minimize the number of other wireless devices active at the same time, our test environment still had considerable interference from devices not in our testbed. With no devices connected to our testbed, the channel utilization ranged between 40% and 50% meaning that roughly half of the channel's airtime was used by interfering traffic or other signals.

To establish a baseline for our later experiments, we sought to find the bit-rate that allowed our video-playback simulation to operate without experiencing network-related errors or performance issues at a 5m distance from the AP. We can control the bit rate of the stream by altering the chunk size (S). To accomplish this, we tested varying chunk sizes from the 5m test location with all devices connected to 3FLCR. The chunk sizes we tested ranged from 187,500 bytes (500 Kbps for a 3-second chunk) to 750,000 bytes (2 Mbps for a 3-second chunk). For each test, we launched our chunk-download simulation on all 20 devices and measured the number of bytes received from the network on each device over a 2-minute window.

We consider error-free operation to be the case where each device is able to keep its buffer from emptying (which would cause user-visible errors, such as buffering). Figure 8.2 shows error-free operation during a chunk-download test. The plot shows the cumulative number of bytes received by the chunk-download simulation over the 120-second test. The black dotted line shows the point at which a device's

Transmit Power Level	Near	Center	Far
High	-37 dBm	-50 dBm	-60 dBm
Medium	-41 dBm	-61 dBm	-67 dBm
Low	-50 dBm	-68 dBm	-75 dBm

Table 8.1: A table of the signal strength at various distances from our testbed access points at various transmit power levels. This shows the expected signal-strength measurements at fixed distances along the line between TEST-2 and TEST-6, which we use to determine where to physically position devices within our testbed in order to inject coverage-gap and sticky-client faults.

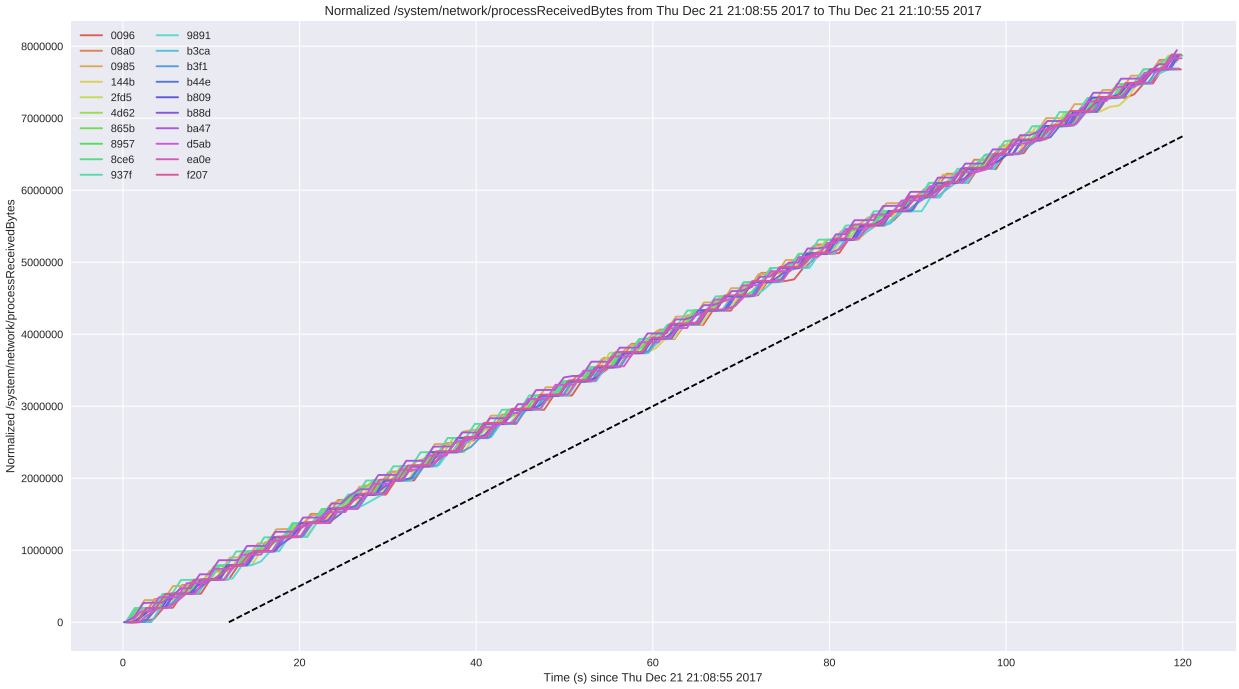

Figure 8.2: A graph showing error-free chunk-download performance, with 20 devices position 5m from the AP downloading 187.5 KB chunks at 4-second intervals. Each solid line represents the number of bytes downloaded from the network over time for a single device. In order to maintain error-free performance, the solid lines must stay above the dotted line. All devices stay above the dotted line throughout the duration of the test.

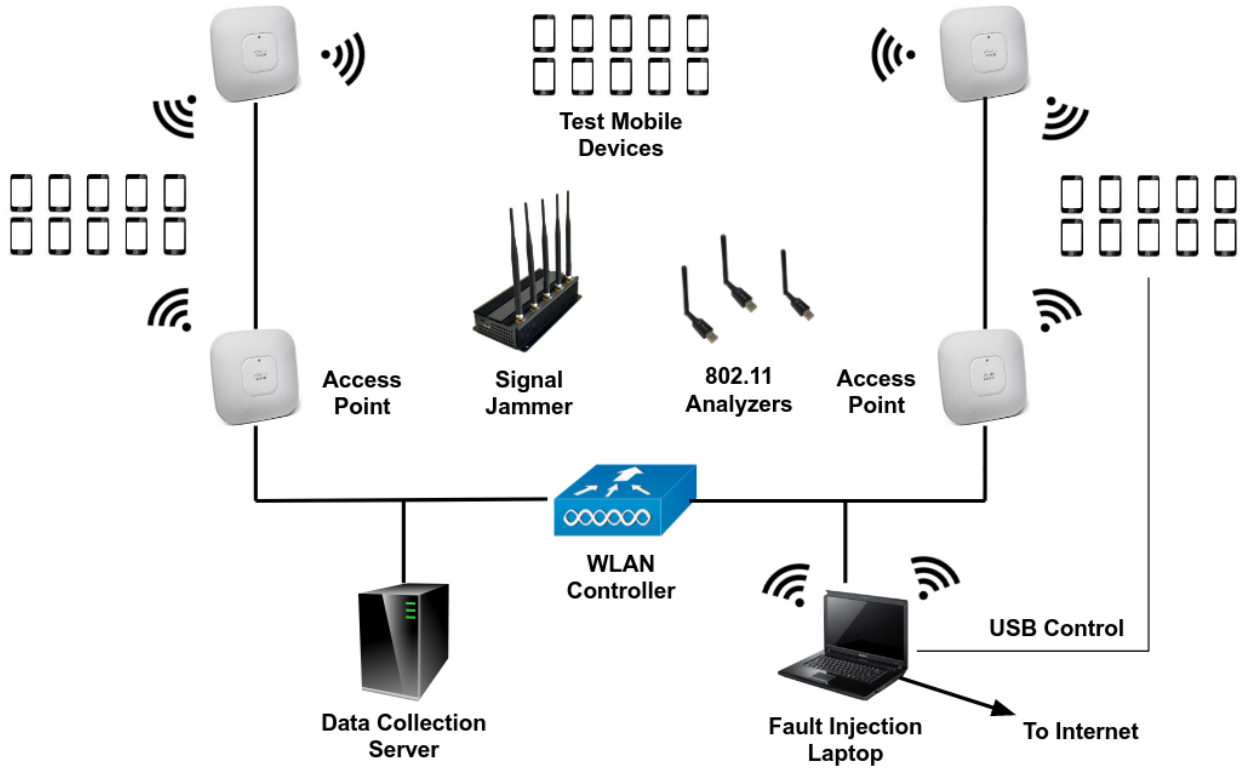

Figure 8.3: A logical diagram of our fault-injection testbed. This diagram shows the equipment used in our testbed in order to create a Wi-Fi network and inject faults into that network. The diagram also shows how the fault-injection and data-collection systems were implemented and connected to the Wi-Fi network.

buffer would be empty. Should one of the devices cross that dotted line, video playback would need to stop for buffering, which we could consider an error. All device buffers remained well above that line, and thus we consider this to be an error-free test.

We ran similar tests with chunks of size 187,500, 375,000, 562,500, and 750,000 bytes (corresponding to 500 Kbps, 1 Mbps, 1.5 Mbps, and 2 Mbps). We found that 20 devices running the simulation with 187,500-byte chunks (500 Kbps) was sufficient to achieve about 80% channel capacity, and larger chunk sizes caused errors even at 5m distances. Based on these results, the remainder of our experiments use 187,500-byte chunks, corresponding to 500 Kbps video streams.

Fault injection. Figure 8.3 shows a diagram of the setup of our fault-injection equipment. Our testbed included the following equipment, allowing us to set up a Wi-Fi network and inject faults:

1. 20x Google Nexus 7 Android Tablets (rooted for control over Wi-Fi network) with instrumented mobile application installed

2. 10x Motorola G5 Smartphones (not rooted) with instrumented mobile application installed

3. 4x Ubiquiti UniFi AP AC PRO access points (unmodified)
4. 1x Ubiquiti Wireless LAN controller (unmodified)
5. 1x 2.4 GHz and 5 GHz signal jammer
6. 3x AirPcap 802.11 packet-capture devices
7. 1x Cisco UCS server with data-collection software installed
8. 1x Lenovo T560 laptop with fault-injection software installed

Our methodology for fault injection is based on the techniques described by Hsueh, Tsai, and Iyer [45]. We use prototype-based fault injection. Our testbed (which includes the prototype system and the fault-injection system) has many of the components described by Hsueh, including a controller and fault injector (running on the T560 laptop), a data-collector (running on the UCS server), and a workload generator (implemented through cooperation of the laptop and the mobile devices). The prototype system in our fault-injection testbed is a Wi-Fi network composed of the four access points, 30 mobile devices, the WLAN controller, and core network services (DHCP, DNS, and NTP) provided by the T560 laptop. The laptop also maintains a route to the Internet for the test Wi-Fi network, which is necessary so that the mobile devices can detect an Internet connection. This is important because if modern Android devices do not detect an Internet connection, they may fail to route packets over the Wi-Fi interface.

While injecting faults, we attempt to avoid direct modification of components as much as possible. Our approach is a combination of the "injection without contact" and software fault-injection approaches defined by Hsueh. For each fault, we configured our testbed to inject that fault by modifying various parameters of the system, such as the number of APs, the signal power and channel assignment on each AP, chunk sizes, and chunk durations (for the synthetic workload). We also use the signal generator to increase the noise floor of our testbed as needed.

Since we aren't directly injecting our faults into the system, we need to verify that we have actually injected our intended faults. To do this, we have multiple ways of monitoring and measuring the RF environment in our testbed. To verify that access points are on the correct channel and the relative differences in signal power between APs at different points in the environment, we use a mobile device with the Wifi Analyzer mobile application [27]. Figure 8.4 shows an example of the visualization that we use to analyze the RF environment, provided by the Wifi Analyzer application. For channel-capacity faults, such as downlink-oversubscription and uplink-contention, we use our AirPcap devices to capture the 802.11 link-layer traffic (frames) once the fault is injected, and verify that the correct fault is injected by inspecting the 802.11 MAC behavior.

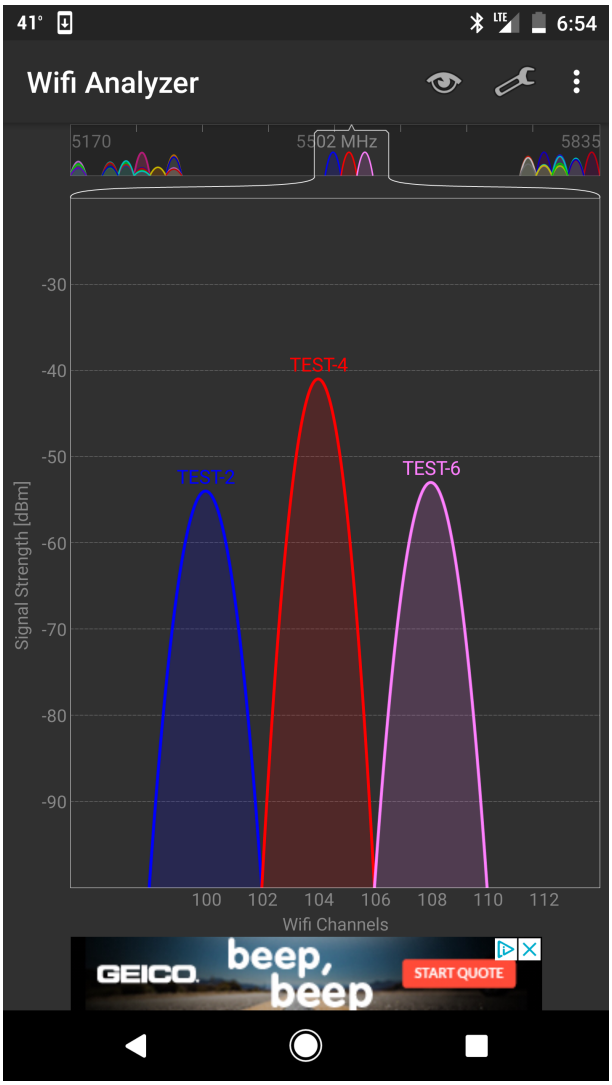

Figure 8.4: A screenshot of the Wi-Fi RF analyzer tool we use in our testbed. The graph shows a real-time analysis of nearby access points, their active channels, and the signal strength to each as measured at the analyzer.

8.2 Diagnosis Performance under Controlled Conditions

Our first set of experiments were designed to evaluate the performance of our diagnosis algorithm in terms of precision and recall for four key faults in our fault model: coverage gap, sticky client, downlink oversubscription, and uplink congestion. We conducted separate experiments for each fault, reconfiguring our testbed to inject the next fault between each experiment, by setting various workload parameters and positioning our devices within the testbed. Once we injected each fault and confirmed that the fault was indeed present on our test devices, we proceeded to collect instrumentation data from devices for 15 minutes. After 15 minutes, we retrieved the devices and downloaded the collected instrumentation data to our data-collection server for offline analysis.

We injected four key performance faults into our testbed using the following methods for each fault.

- **Coverage gap.** To inject the coverage-gap fault, we used two of the access-points, TEST-4 and TEST-6, and disabled TEST-2. Both active access points used low signal power. We then placed the devices to experience the fault in the Far position from TEST-6 (near TEST-2). We confirmed experimentally that, in this position, these devices were not able to reliably execute the video-playback workload while associated either TEST-4 or TEST-6.

- **Sticky client.** For the sticky-client fault, we activated all three access points. We then forced the devices under test to associate with TEST-2 (using a unique SSID) and placed them in the Far position from TEST-2. We then experimentally verified that the devices were unable to execute the video-playback workload while associated to either TEST-4 or TEST-6.

- **Downlink oversubscription.** To inject the downlink-oversubscription fault, we activated all access points and placed the devices under test in the Near position to TEST-6. We then configured the devices under test to use 4.5-megabyte chunks downloaded at 4-second intervals. With 9 devices running this workload, the the throughput required exceeds the airtime available. We confirmed this using our AirPCap monitoring, noting that our monitoring tool measured 100% channel utilization while the retransmission rate remained at 15% (the baseline typically seen under fault-free conditions).

- **Uplink contention.** Similar to downlink oversubscription, we injected this fault by activating all access points and placing the devices under test in the Near position to TEST-6. We again used 9 test devices, with 6 devices running a chunk-upload workload (the same as chunk-download, but uploading chunks from the device to the server) using 3.75-megabyte chunks uploaded at 1-second intervals. The remaining three devices attempted the standard chunk-download workload using

Fault	Precision	Recall
Coverage Gap	1.000	0.818
Sticky Client	1.000	1.000
Downlink Oversubscription	0.988	0.988
Uplink Contention	1.000	1.000

Table 8.2: Performance of our problem-diagnosis algorithm under controlled conditions.

750-kilobyte chunks at 4-second intervals. Under these conditions, the airtime utilized by the upload clients prevents the download clients from obtaining enough throughput for their workloads. We verified this using our AirPCap monitoring, noting that our monitoring tool measured 100% channel utilization with the retransmission rate climbing to 25% as a result of the upload devices interfering with each other's transmissions.

Once we injected and confirmed each fault (with our synthetic workload also active), we collected our instrumentation data from that fault for a period of 15 minutes. This data was collected to the local storage on each device, rather than being sent to the data-collection server, to avoid interfering with the experiment. The instrumentation data included all of the metrics we discussed, along with the output of our real-time problem detection. After the 15-minute period had elapsed, we collected all of our test devices and downloaded our logs onto the data-collection server. Once collected, we scanned each log and, for each detected fault, we ran our diagnosis engine to diagnose that fault, recording the result. If the diagnosed fault matched the injected fault, we recorded a true positive. Otherwise, we recorded a false positive for the fault that was diagnosed and a false negative for the fault being injected.

Table 8.2 shows the results of our experiment. We found that our diagnosis engine was able to diagnose all of the faults in our fault model with between 0.988 to 1.000 precision and 0.818 to 1.000 recall. Excluding the one fault (coverage gap) with 0.818 recall, the remaining faults had between 0.988 and 1.000 recall as well. We attribute the 0.818 recall for coverage gap to high variance in measured RSSI at long distances from the access point. If we were able to space our access points further in the test environment, the coverage gap fault likely would have had better recall performance.

8.3 Diagnosis Performance in Real-World Environments

Having studied the effectiveness of our problem diagnosis on a prototype system in a laboratory environment, we wanted to take our approach into a real-world setting and determine if it could be applied in that environment as well. To do this, we searched for a real-world location that would replicate the

conditions found in production Wi-Fi environments, in which we could set up our test equipment. The UPMC Lemieux Sports Complex (LSC) in Cranberry, PA was kind enough to allow us to perform our experiments at their facility.

The LSC is the official training facility for the Pittsburgh Penguins NHL team, and contains two regulation-size hockey rinks with spectator seating for about 1000 fans at each rink. Outside of Penguins practices, the rink is frequently used for training at other levels of play, and frequently hosts games for a number of other hockey leagues, including the AAA minor ice hockey team Penguins Elite. We were fortunate to be able to use the facility for experiments during the Mid-Am AAA District Playoffs from March 8 through 11, 2018. This event involved eight minor ice hockey teams from the states of Ohio, Pennsylvania, and West Virginia, playing 18 hockey games over the 4-day period.

As we were unable to modify the Wi-Fi network in the building for testing (as this would disrupt the Wi-Fi performance for event attendees), we deployed our testbed in a real-world configuration on Rink #1 of the LSC. Similar to a hockey arena, the spectator area is divided into seven seating sections, each seating between 100 and 150 spectators at maximum capacity. Each game played during the playoffs regularly drew between 300 and 400 fans, most of which were seated in the middle 3 sections (3, 4, and 5) with a smaller number in sections 2 and 6 and very few in sections 1 and 7.

We deployed our Wi-Fi system by placing three access points in the rear of sections 2, 4, and 6, providing Wi-Fi coverage to those sections. The access points were spaced approximately 16 meters apart. We set up our fault injection equipment and workload generator in the rear of section 4 near the access point, on a table designated for press. This area had wired Internet connectivity, which provided our network with Internet access. We installed a PoE-enabled Cisco switch at this location and ran Ethernet cables from the switch to the APs in sections 2 and 4 to provide the APs with power and data connections. Our APs were set to use channels 100, 104, and 108, which are non-overlapping channels in the DFS (dynamic frequency selection) range. We chose these channels as no other APs in the facility used these channels, so this ensured we had clean Wi-Fi channels with which to conduct our experiments without interfering with the Wi-Fi service in the facility. While these channels are in the DFS range, there were no radar systems in operation nearby on the 5 GHz band, and thus the DFS function was not triggered during our experiments.

We conducted our experiments in the same manner as our experiments in the lab. We injected faults and again verified that we had injected those faults successfully in the new environment. We injected our faults during times when games were being played to gather data under real-world conditions, where spectators are in the space and move around throughout the game. We collected data under each fault condition for at least 20 minutes under these conditions, storing the logs to local storage and then collect-

Fault	Precision	Recall
Coverage Gap	0.977	0.982
Sticky Client	1.000	0.896
Downlink Oversubscription	0.980	0.997

Table 8.3: Performance of our problem-diagnosis algorithm under real-world conditions.

ing them onto our fault-injection laptop (now also serving as the data-collection server) following each experiment.

To measure the performance of our problem-diagnosis engine in this real-world setting, we ran our problem detector and diagnosis engine on the collected logs and measured false positives and true positives, as we had done in the lab. Our results are presented in Table 8.3.

In a real-world setting, we found that our diagnosis engine was able to diagnose coverage gap, sticky client, and downlink oversubscription with between 0.977 to 1.000 precision and 0.896 and 0.997 recall. We found that, despite the increased noise from the motion of objects within the space, our approach improved in performance on the coverage-gap fault because there was more distance between the two furthest access points (32 meters versus 25 meters in our lab configuration). However, the noise in our measurements created by motion throughout the space took its toll on the other types of faults, with only sticky client achieving perfect 1.000 precision, but having considerably poorer recall than in the lab.

8.4 Effectiveness of Mitigation

Finally, we evaluate the effectiveness of our automated-mitigation strategy. To do this, we follow a similar lab setup as in previous experiments, injecting faults to which automated mitigation can be applied (sticky-client, downlink-oversubscription, and uplink-congestion). For each fault, we configured our diagnosis engine to immediately apply our performance-aware AP-selection algorithm once a problem is detected, rather than attempt to diagnose the problem. Because we need to apply the mitigation in real time as the fault is being injected to measure its effectiveness, we implemented and tested an online version of our problem-diagnosis and problem-mitigation algorithms.

To test our problem-mitigation approach, we placed our nine test devices near TEST-4 in our testbed and connected three devices each to TEST-2, TEST-4, and TEST-6. The three devices connected to TEST-2 experienced a fault during our tests, while TEST-4 was heavily loaded by another set of three devices and TEST-6 was lightly loaded by the last set of three devices. We configured the load in this manner because the simple strongest-signal-strength approach would select TEST-4 as the best access point due to signal

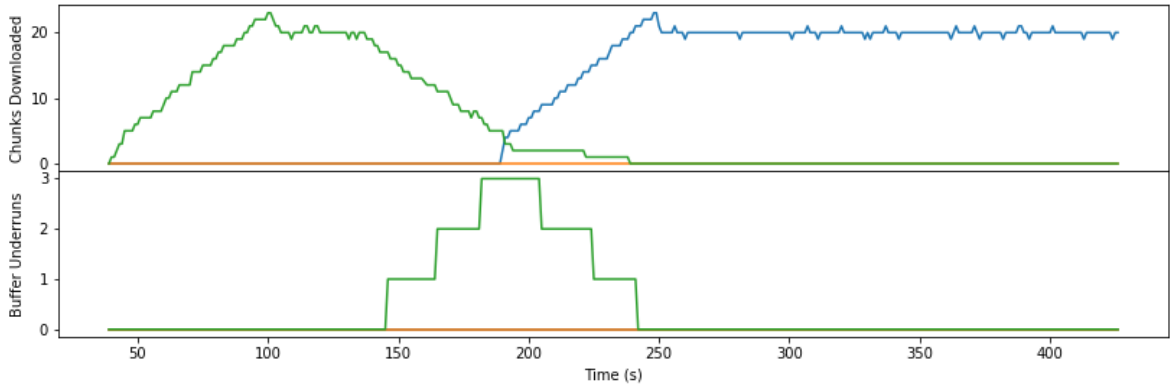

Figure 8.5: This pair of graphs shows the effectiveness of our mitigation strategy. The x-axis indicates the time from the start of our experiment. The upper graph plots the number of chunks downloaded within the last 60 seconds at 5-second intervals. The lower graph plots the number of errors (buffer underruns) that occurred over the last 60 seconds at 5-second intervals. Between 130s and 185s, a fault is present resulting in a decrease in chunks downloaded and an increase in errors. At 185s, our mitigation strategy forces the device to connect to a different access point, resulting in restored chunk downloads (the second line on the upper graph), and the number of errors decreasing to 0 for the remainder of the experiment.

strength, but in fact TEST-6 offers better performance due to load. Our goal, in each test, was for all three devices on TEST-2 to detect a fault and mitigate the problem by associating with TEST-6 (not TEST-4).

For each test run, we configured the nine test devices to connect to their assigned access points and begin running a workload. For TEST-2, the workload was downloading 1.5-megabyte chunks every 4 seconds. For TEST-4, the workload was downloading 4.5-megabyte chunks every 2 seconds. For TEST-6, the workload was downloading 375-kilobyte chunks every 4 seconds. The workload on TEST-2 was sufficient to create faulty conditions that could be diagnosed as either sticky client or downlink oversubscription, as there was both an access point offering much higher signal strength nearby (TEST-4), and this workload consumed nearly 100% of the airtime on the channel assigned to TEST-2.

Figure 8.5 shows an example of our problem-mitigation algorithm in action on a single test device. The x-axis indicates the time since the start of the experiment, and the two graphs indicate the number of chunks downloaded over the last 60 seconds (upper) and the number of buffer-underruns that have occurred in the last 60 seconds (lower). An increase in the number of buffer underruns indicates a user-visible performance problem. The colors in the graph indicate which of the three access points were active (associated) when the event occurred (a chunk was downloaded or a buffer-underrun event occurred).

The graph shows normal, error-free playback from $t = 0$ to approximately $t = 130$. Note that the number of chunks downloaded increases linearly with no buffer underruns through $t = 90$, and then levels off once chunk-download events begin falling out of the 60-second sliding window. We injected the fault at approximately $t = 130$, and the number of chunks downloaded begins falling shortly thereafter.

This results in a buffer underrun at $t = 145$, triggering our problem diagnosis and mitigation system. Between $t = 145$ and $t = 185$ our performance-aware mitigation strategy collects instrumentation data from nearby peers using BLE. Finally, at $t = 185$, the mitigation algorithm has enough data from peers to decide that TEST-6 is the best access point and forces the device to associate with TEST-6. The device immediately begins downloading chunks again once the association with TEST-6 completes, and there are no further buffer-underrun events after this point in the run.

We note that the latency from detection to mitigation is approximately 40 seconds in this case and in our experimental results is often longer than this. This latency is due to the unreliable nature of BLE communication. While we found that BLE GATT operations are reliable for a small number of devices (2 to 3) communicating with each other, as this number grows to 9-way peer-to-peer communication, connections often fail and operations frequently time out. The root cause of this problem is unknown. However, based on our attempts to resolve the problem, we believe the issue may be due to bugs or limitations in the BLE drivers on Android (we experienced this problem across Android devices from three different manufacturers). We worked around this problem by imposing short timeouts on connection and read operations, and retrying these operations several times before giving up. Often, a connection or read attempt will succeed the third or fourth time it is attempted. While this workaround provides robustness to our approach, the necessity of these retries results in increased latency for the overall mitigation operation.

Our mitigation approach successfully selected a better access point in 100% of our trials, resulting in the resumption of error-free execution of our workload. Also, in all trials, the access point with the lightest load (TEST-6) was chosen over the closer access point (TEST-4). Latency from the initial user-visible problem to the last occurrence of a user-visible problem ranged from 40 seconds to 140 seconds, with a mean delay of 94 seconds.

Figure 8.6 shows an example of a phenomenon we experienced when we allowed all nine devices to perform problem mitigation (not just the three designated to experience the fault). We allowed our system to run for an extended period of time (15 minutes) and observed the results. The single-device trace shows a mitigation attempt at approximately 200 seconds into the experiment, switching from TEST-2 to TEST-4. Because buffer underruns continue to occur, the device switches again from TEST-4 to TEST-6. After some time, other devices also switch to TEST-6, eventually overloading that access point and triggering additional mitigation actions at 240s, 550s, 625s, 710s, and 725s. The system eventually stabilizes after about 12 minutes.

This example shows cascading failures triggered by our mitigation actions. It indicates that more work needs to be done to prevent performance-aware problem-mitigation from inadvertently causing additional problems by overloading access points that were previously not loaded or only lightly loaded. It may be

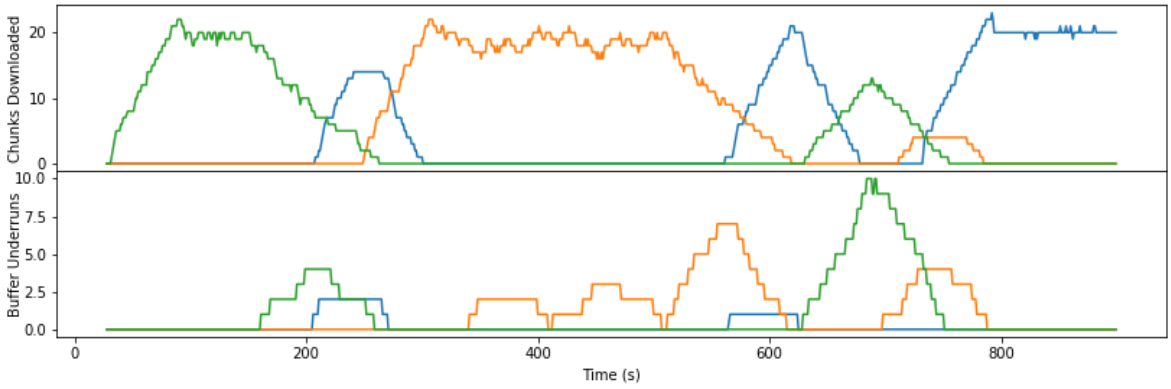

Figure 8.6: This pair of graphs shows an example of cascading failures triggered by our problem-mitigation strategy. The x-axis indicates the time from the start of our experiment. The upper graph plots the number of chunks downloaded within the last 60 seconds at 5-second intervals. The lower graph plots the number of errors (buffer underruns) that occurred over the last 60 seconds at 5-second intervals. The graph shows 7 mitigation attempts at 200s, 240s, 550s, 625s, 710s, and 725s. All mitigation attempts beyond the first were triggered by overloading of the alternate access point caused by several devices associating with that access point while attempting to mitigate a problem. The system finally stabilizes after nearly 12 minutes.

possible to avoid these failures by limiting the number of mitigation actions or coordinating among peers to avoid all peers simultaneously applying the same mitigation strategy.

Chapter 9

Conclusion and Future Work

9.1 Summary of Dissertation

Large-scale, high-density Wi-Fi networks are becoming increasingly common, being deployed in stadiums, arenas, schools, airports, and other large public spaces. These networks use hundreds of access points to serve thousands of simultaneous clients. These systems are complex, composed of multiple layers with each layer containing multiple components. They are also dynamic, as the performance characteristics of the network change with time, as users and other objects enter, move through, and eventually exit the physical space, altering it's RF characteristics and the workload placed on the network.

Users of large-scale, high-density Wi-Fi networks expect to be able to use applications that demand both low latency and high bandwidth, such as live-video streaming. These users expect such applications to function without error at any location in the space, regardless of the number or density of other users nearby. This poses a challenge to designers and operators of these Wi-Fi networks, as not only does the network need to provide adequate signal coverage throughout the entire physical space, it also needs to provide error-free low-latency, high-bandwidth services to all users of the network even under peak load.

To meet this challenge, we created a problem-detection, problem-diagnosis, and automated-mitigation system for production large-scale, high-density Wi-Fi networks. Our goal was to create such a system that detected user-visible problems in near-real-time (within seconds), diagnosed these problems with low cost (in terms of invasiveness of the approach and financial cost of deployment and maintenance), and automatically mitigated faults that occur. To guide the construction of our system, we designed, implemented, and deployed the first system to instrument a real-world production live-video-streaming application to collect client-side Wi-Fi performance data from real-world production large-scale, high-density Wi-Fi networks. Using the data we collected from thousands of devices at streaming live video at 25 stadiums over 3 years, we determined how Wi-Fi performance problems impact user behavior,

providing insight into when an error in video playback becomes a user-visible performance problem. With this insight in hand, we proceeded to develop a real-time problem-detection system based on our client-side instrumentation data. Using our knowledge of the construction of production large-scale, high-density Wi-Fi networks and the work of other researchers, we constructed a comprehensive fault model covering common faults across all layers of the Wi-Fi network. Based on this this fault model, we designed a novel problem-diagnosis algorithm that uses multiple perspectives of the network to diagnose the root cause of detected problems. We then developed an automated-mitigation system to attempt to mitigate problems based on the diagnosed fault, if that fault could be mitigated by switching to an alternate access point. Finally, we validated our approach through both laboratory and real-world testing, showing that our problem-diagnosis approach is able to diagnose root cause with high precision and recall in both lab and real-world environments, and that our automated-mitigation approach successfully mitigates Wi-Fi performance problems within minutes of problem detection.

9.2 Open Questions and Future Work

Instrumentation of additional operating systems and Wi-Fi infrastructure from additional vendors. Our approach today is only compatible with the Android operating system and Cisco Wi-Fi infrastructure. This excludes other smartphone operating systems, most notably Apple's iOS, as well as Wi-Fi infrastructure from other vendors. To be broadly applicable to all production Wi-Fi networks, our approach would need to be extended to include iOS and other Wi-Fi infrastructure vendors.

Use of infrastructure-side data in problem diagnosis. Our current diagnostic approach does not use data from the Wi-Fi infrastructure because the instrumentation data from the infrastructure is updated too infrequently (by the infrastructure) to be correlated with our real-time client-side instrumentation data. It may be possible to obtain more frequent updates from Cisco infrastructure, or to obtain high-frequency data from equipment from other Wi-Fi infrastructure vendors. Using this data, it may be possible to diagnose a wider range of faults, or diagnose faults more accurately than is possible from multiple client-side perspectives alone.

Diagnosis of faults across all layers of our fault model. Currently, our diagnosis approach focuses on six faults that occur at three of the six layers of the Wi-Fi network. There are many other faults that may occur across all six layers. Our problem-diagnosis approach could be expanded to cover this wider range of faults.

Mitigation of a wider range of faults. Currently, our automated-mitigation system only attempts mitigation for faults that can be mitigated through alternate-access-point selection. Other faults can be mitigated

by physically moving the device, or perhaps other automated techniques such as resetting the Wi-Fi driver or updating Wi-Fi configuration. These mitigation strategies could be implemented to extend the capabilities of our automated-mitigation system.

Problem diagnosis and mitigation with seconds of latency. Currently, our problem-diagnosis algorithm (and subsequent automated mitigation) takes on the order of minutes to execute due to the time required to communicate with nearby peers and obtain performance data from their perspectives using Bluetooth. This could be improved, perhaps through the use of a different protocol or by fixing the limitations of the Bluetooth subsystem on Android to permit fast communication with a large number of peers. Ideally, this process would complete before the 6-second or 12-second threshold where users disengage, thus preventing an error from becoming a user-visible problem.

Instrumentation of a wider range of smartphone applications. Currently, our instrumentation system only obtains performance metrics from the Android video player. This could be extended to other types of applications (e.g. photo sharing, web browsing, etc.). It also may be possible to develop a single approach that is applicable to many applications by automatically learning the typical error-free network behavior for an application (perhaps by analyzing network traffic patterns) and then determining the correlation between atypical patterns and user-visible errors.

Automated discovery of parameters for our problem-diagnosis algorithm. Our problem-diagnosis algorithm relies on several threshold values. The thresholds that optimize the performance of our problem-diagnosis approach vary between Wi-Fi networks. These values can be obtained through empirical testing and analysis of Wi-Fi performance data from the network under study, using fault injection and data collection. We have used this approach in this dissertation with success, but it may be possible to automatically learn the correct thresholds for any Wi-Fi network without going through this process.

Keeping users engaged through user interaction. Our studies of real-world user behavior provided the insight that users disengage from our live-video-streaming application quickly when faced with Wi-Fi performance problems. Our system currently diagnoses and mitigates these problems behind the scenes, without any interaction with the user. During this time, the user may disengage anyway, not knowing that the problem is in the process of being resolved. To keep the user engaged with the application while diagnosis/mitigation is occurring, it may be possible to develop a graphical user interface around our diagnosis system to keep the user informed during this process.

Network-wide visualization of Wi-Fi performance problems from the client's perspective. In addition to end users, it is also important that network operators know when problems occur, their root causes, and what is being done (if anything) to mitigate the problem. This allows operators to track the performance of the system and perhaps send Wi-Fi technicians or customer-service representatives to locations where

there is a high incidence of performance problems, in order to assist guests experiencing those problems. It may be possible to construct such a network-wide visualization of these problems based on the Wi-Fi performance data that we collect.

Avoiding cascading failures due to automated mitigation. As discussed in our evaluation, it is possible for our automated-mitigation strategy of alternate-access-point selection to cause cascading failures if many devices simultaneously detect a problem and choose to associate with another access point. If the workload produced by all of those devices in aggregate is the cause of the problem, shifting all of those devices to another (single) access point will just shift the location of the problem (and perhaps make the problem worse if other devices are now impacted). There may be algorithmic techniques that can protect the system from these cascading failures, for example through coordination among peers to ensure that only a small number of peers apply mitigation at the same time. It may be possible to compute an optimal network-wide assignment of devices to access points based on the physical location of the device and our collected Wi-Fi performance data.

Bibliography

[1] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and techniques for diagnosing faults in ieee 802.11 infrastructure networks. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 30–44. ACM, 2004.

[2] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11 b mesh network. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 121–132. ACM, 2004.

[3] N. AiroPeek. Wildpackets. *Inc. http://www. wildpackets. com*, 2004.

[4] A. Akella, G. Judd, S. Seshan, and P. Steenkiste. Self-management in chaotic wireless deployments. *Wireless Networks*, 13(6):737–755, 2007.

[5] Apple. AVPlayer. https://developer.apple.com/documentation/avfoundation/avplayer, 2018. [Online; accessed 14-Mar-2018].

[6] Apple. HTTP Live Streaming. https://developer.apple.com/streaming/, 2018. [Online; accessed 14-Mar-2018].

[7] P. Bahl, R. Chandra, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. Enhancing the security of corporate wi-fi networks using dair. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 1–14. ACM, 2006.

[8] A. Balachandran, G. M. Voelker, P. Bahl, and P. V. Rangan. Characterizing user behavior and network performance in a public wireless lan. *SIGMETRICS Perform. Eval. Rev.*, 30(1):195–205, June 2002.

[9] M. Balazinska and P. Castro. Characterizing mobility and network usage in a corporate wireless local-area network. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, MobiSys '03, pages 303–316, New York, NY, USA, 2003. ACM.

[10] D. Barrera. Wifighter: Improving access point selection under linux. 2008.

[11] S. Biswas, J. Bicket, E. Wong, R. Musaloiu-e, A. Bhartia, and D. Aguayo. Large-scale measurements of wireless network behavior. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 153–165. ACM, 2015.

[12] A. Blog. WiFi Network Design Considerations. https://www.accessagility.com/blog/wifi-network-design-considerations, 2018. [Online; accessed 4-Apr-2018].

[13] A. Carroll, G. Heiser, et al. An analysis of power consumption in a smartphone.

[14] R. Chandra, V. N. Padmanabhan, and M. Zhang. Wifiprofiler: cooperative diagnosis in wireless lans. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 205–219. ACM, 2006.

[15] X. Chen, B. Wang, K. Suh, and W. Wei. Passive online wireless lan health monitoring from a single measurement point. *ACM SIGMOBILE Mobile Computing and Communications Review*, 14(4):19–21, 2011.

[16] Y.-C. Cheng. *Automating cross-layer diagnosis of enterprise 802.11 wireless networks*. ProQuest, 2007.

[17] Y.-C. Cheng, M. Afanasyev, P. Verkaik, P. Benko, J. Chiang, A. C. Snoeren, S. Savage, and G. M. Voelker. *Shaman Automatic 802.11 Wireless Diagnosis System*. [Department of Computer Science and Engineering], University of California, San Diego, 2010.

[18] Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage. *Jigsaw: solving the puzzle of enterprise 802.11 analysis*, volume 36. ACM, 2006.

[19] Cisco. Cisco Prime Infrastructure – Programming Guides. https://www.cisco.com/c/en/us/support/cloud-systems-management/prime-infrastructure/products-programming-reference-guides-list.html, 2018. [Online; accessed 14-Mar-2018].

[20] Cisco Systems. *Cisco Prime Infrastructure*.

[21] Cisco Systems. *CMX*.

[22] S. Das and V. Kone. Retimon–a real time network monitor.

[23] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 362–373. ACM, 2011.

[24] D. Eckhardt and P. Steenkiste. Measurement and analysis of the error characteristics of an in-building wireless network. In *ACM SIGCOMM Computer communication review*, volume 26, pages 243–254. ACM, 1996.

[25] T. Elomaa and J. Rousu. Finding optimal multi-splits for numerical attributes in decision tree learning. *ESPRIT Working Group, NeuroCOLT Technical Report Series*, pages 1–16, 1996.

[26] M. Ergen. IEEE 802.11 Tutorial. http://www-inst.eecs.berkeley.edu/~ee228a/fa03/228A03/802.11%20wlan/802.11_tutorial.pdf, 2018. [Online; accessed 6-Apr-2018].

[27] farproc. Wifi Analyzer. https://play.google.com/store/apps/details?id=com.farproc.wifi.analyzer&hl=en, 2018. [Online; accessed 16-Mar-2018].

[28] A. Farshad, J. Li, M. K. Marina, and F. J. Garcia. A microscopic look at wifi fingerprinting for indoor mobile phone localization in diverse environments. In *Indoor Positioning and Indoor Navigation (IPIN), 2013 International Conference on*, pages 1–10. IEEE, 2013.

[29] U. Fayyad and K. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. 1993.

[30] D. I. Forum. DASH Industry Forum. http://dashif.org/, 2018. [Online; accessed 14-Mar-2018].

[31] T. E. Foundation. jetty://. https://www.eclipse.org/jetty/, 2018. [Online; accessed 14-Mar-2018].

[32] J. Geier. How to: Conduct a Wireless Site Survey. http://www.wireless-nets.com/resources/tutorials/conduct_wireless_site_survey.html, 2018. [Online; accessed 4-Apr-2018].

[33] C. Gomez, J. Oller, and J. Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[34] Google. android.net.wifi. https://developer.android.com/reference/android/net/wifi/package-summary.html, 2018. [Online; accessed 14-Mar-2018].

[35] Google. Bluetooth Low Energy. https://developer.android.com/guide/topics/connectivity/bluetooth-le.html, 2018. [Online; accessed 6-Apr-2018].

[36] Google. Media Player. https://developer.android.com/reference/android/media/MediaPlayer.html, 2018. [Online; accessed 14-Mar-2018].

[37] Google. TelephonyManager. https://developer.android.com/reference/android/telephony/TelephonyManager.html, 2018. [Online; accessed 14-Mar-2018].

[38] Google. VideoView. https://developer.android.com/reference/android/widget/VideoView.html, 2018. [Online; accessed 14-Mar-2018].

[39] Google. WifiScanner. https://android.googlesource.com/platform/frameworks/base/+/android-p-preview-1/wifi/java/android/net/wifi/WifiScanner.java, 2018. [Online; accessed 14-Mar-2018].

[40] T. Henderson, D. Kotz, and I. Abyzov. The changing usage of a mature campus-wide wireless network. In *Proceedings of the 10th annual international conference on Mobile computing and networking*, MobiCom '04, pages 187–201, New York, NY, USA, 2004. ACM.

[41] S. D. Hermann, M. Emmelmann, O. Belaifa, and A. Wolisz. Investigation of ieee 802.11 k-based access point coverage area and neighbor discovery. In *Local Computer Networks, 2007. LCN 2007. 32nd IEEE Conference on*, pages 949–954. IEEE, 2007.

[42] K. Heurtefeux and F. Valois. Is rssi a good choice for localization in wireless sensor network? In *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, pages 732–739. IEEE, 2012.

[43] M. Heusse, F. Rousseau, G. Berger-Sabbatel, and A. Duda. Performance anomaly of 802.11b. In *Proceedings of IEEE Infocom 2003*, 2003.

[44] L. HowTos. /proc/stat explainedr. http://www.linuxhowtos.org/System/procstat.htm, 2018. [Online; accessed 14-Mar-2018].

[45] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[46] T. iPhone Wiki. Private Frameworks. https://www.theiphonewiki.com/wiki//System/Library/PrivateFrameworks, 2018. [Online; accessed 14-Mar-2018].

[47] H. Kim, W. Lee, M. Bae, and H. Kim. Wi-fi seeker: a link and load aware ap selection algorithm. *IEEE Transactions on Mobile Computing*, 16(8):2366–2378, 2017.

[48] K.-H. Kim, H. Nam, and H. Schulzrinne. Wislow: A wi-fi network performance troubleshooting tool for end users. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 862–870. IEEE, 2014.

[49] M. Klepal, S. Zvanovec, and P. Pechac. Wireless lan networks design: Site survey or propagation modeling? *Radio Engineering*, 12(4), Dec. 2003.

[50] A. Mahanti, C. Williamson, and M. Arlitt. Remote analysis of a distributed wlan using passive wireless-side measurement. *Performance Evaluation*, 64(9):909–932, 2007.

[51] R. A. Maxion and R. T. Olszewski. Detection and discrimination of injected network faults. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 198–207. IEEE, 1993.

[52] N. D. Mickulicz, U. Drolia, P. Narasimhan, and R. Gandhi. Zephyr: First-person wireless analytics from high-density in-stadium deployments. In *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–10. IEEE, 2016.

[53] N. D. Mickulicz, P. Narasimhan, and R. Gandhi. YinzCam: Experiences with in-venue mobile video and replays. In *USENIX LISA*, pages 133–144, Nov. 2013.

[54] Mobile Sports Report. *State of the Stadium Technology Survey 2015*.

[55] R. Murty, J. Padhye, A. Wolman, and M. Welsh. Dyson: An architecture for extensible wireless lans. In *USENIX Annual Technical Conference*, 2010.

[56] A. J. Nicholson, Y. Chawathe, M. Y. Chen, B. D. Noble, and D. Wetherall. Improved access point selection. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 233–245. ACM, 2006.

[57] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[58] A. Patro, S. Govindan, and S. Banerjee. Observing home wireless experience through wifi aps. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 339–350. ACM, 2013.

[59] C. Pei, Y. Zhao, G. Chen, R. Tang, Y. Meng, M. Ma, K. Ling, and D. Pei. Wifi can be the weakest link of round trip network latency in the wild.

[60] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. 2010.

[61] L. Qiu, P. Bahl, A. Rao, and L. Zhou. Fault detection, isolation, and diagnosis in multihop wireless networks. Technical report, Technical Report MSR-TR-2004-11, Microsoft Research, Redmond, WA, 2003.

[62] R. Raghavendra, P. A. K. Acharya, E. M. Belding, and K. C. Almeroth. Antler: A multi-tiered approach to automated wireless network management. In *INFOCOM Workshops 2008, IEEE*, pages 1–6. IEEE, 2008.

[63] P. Rajamani. Application Visibility and Control âĂŞ Managing AVC with Cisco Prime Infrastructure 2.0. http://d2zmdbbm9feqrf.cloudfront.net/2013/usa/pdf/BRKNMS-1040.pdf, 2018. [Online; accessed 6-Apr-2018].

[64] roman10. How to get CPU information on Android. http://www.roman10.net/2011/12/31/how-to-get-cpu-information-on-android/, 2018. [Online; accessed 14-Mar-2018].

[65] Y. Sheng, G. Chen, H. Yin, K. Tan, U. Deshpande, B. Vance, D. Kotz, A. T. Campbell, C. McDonald, T. Henderson, et al. Map: a scalable monitoring system for dependable 802.11 wireless networks. *IEEE Wireless Commun.*, 15(5):10–18, 2008.

[66] A. Sheth, C. Doerr, D. Grunwald, R. Han, and D. Sicker. Mojo: A distributed physical layer anomaly detection system for 802.11 wlans. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 191–204. ACM, 2006.

[67] J. Shi. How Android Wi-Fi State Machine Works. http://jhshi.me/2014/04/25/how-android-wifi-state-machine-works/index.html#.Wqlg-3XytQM, 2014. [Online; accessed 14-Mar-2018].

[68] J. Shi, L. Meng, A. Striegel, C. Qiao, D. Koutsonikolas, and G. Challen. A walk on the client side: Monitoring enterprise wifi networks using smartphone channel scans. *Proceedings of INFOCOMâĂŹ16*, 2016.

[69] R. K. Singh and N. Tiwari. An investigation on wireless mobile network and wireless lan (wi-fi) for performance evaluation. *International Journal of Computer Applications*, 126(6), 2015.

[70] StatCounter. Mobile & Tablet Operating System Market Share in North America - February 2018. http://gs.statcounter.com/os-market-share/mobile-tablet/north-america/#monthly-201702-201802, 2018. [Online; accessed 14-Mar-2018].

[71] StatCounter. Mobile & Tablet Operating System Market Share Worldwide - February 2018. http://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-201702-201802, 2018. [Online; accessed 14-Mar-2018].