MASTER'S THESIS

A Generic Event Logger For Real-Time, Web-Based Applications

Timelines, Selection & Intelligent Replay

Supervisor: Prof. Dr. Anton Eliens

Leonidas Diamantis
l.diamantis@student.vu.nl
VU University 2014

Acknowledgments

This document is the result of combined research, considerations and efforts to create a system that uses technology to produce multi-dimensional results, from analytical insights to artistic performances. A system that allows users to “use the past in order to change the future”. During the implementation process there were many scientific, technical and moral aspects of the topic that account for, and quite a few decisions - both conceptual and technical - had to be made. And I honestly feel very lucky that I had a number of people who helped me in so many ways throughout this so challenging endeavor.

First and foremost, I would like to thank my supervisor, Prof. Dr. Anton Eliens, for his guidance & support, for all the times he inspired me to continue and all patience he showed with my technical dilemmas. I would also like to express my gratitude to Prof. Dr. Spyros Voulgaris for being an excellent academic mentor for me, for agreeing to be the second reader of this document and for the invaluable advice he always offered, especially during the kickstart and the design phase of my Master’s Thesis. Furthermore I would like to thank my friends and colleagues Chronis Kalaitsidis, Andreas Manios & Nikos Poulios for sharing their comments and ideas with me during our conversations about this project. Last but not least, I want to thank Konstantina Mavridou for believing in me and for dedicating so much of her time & energy to morally support me along the way. I will forever be grateful for this.

Chapter 1: Introduction

The concept of posterior analysis of system behaviors is essential in the world of science, inspiring and concerning multiple fields from Mathematics to Computer Science and Business Intelligence, stimulating people to consider & invent advanced techniques and methods which would help them analyze the behavior of certain systems and draw useful conclusions about how they (mis)behave.

The whole process of such analysis boils down to one main task: extracting information from a dataset with data specific to the system's execution, and performing different kinds of analyses to determine how that system behaved, based on some criteria. Obviously, the nature of this dataset as in what types of data it carries will produce different kinds of conclusions at different depths. With the most common approach (from a Computer Science perspective) on this matter being that such information set would mainly include raw, computationally produced data (e.g the sum of two numbers for an addition application) the conclusion-making process is bound to follow specific analysis patterns and produce conclusions following respective patterns as well.

The system built for this thesis comes to offer an alternative approach to the one described above, in respect to the creation of the dataset: for our scope, we focus on capturing the actual interaction between user and application and creating a dataset which, instead of the data that the application produces, consists of meta-information about how has the application been made use of, from the user's standpoint. And with the (selective) replay features our system implements, the conclusion-making process for the application's behavior could be done by examining exactly what kind of usage made the system behave in the way it did.

Motivations

The research carried out for this Thesis was motivated by two main domains: computational analysis & the digital arts.

While considering the scope & approach for this project, we considered various use-case scenarios and types of systems which would render such a logging system useful. We considered collaborative applications which impose distortions in the communication between participants and various distributed systems which have the tendency to fail on arbitrary moments and modes. For such scenarios, we thought that such a system would make “the building blocks of determining what went wrong in a distributed system”, from a

user's perspective, as the information in which the logger is interested is the actual usage that made the application behave in the (possibly faulty) way that it did.

As a counterpart to the strict, Computer Science driven domain, we also considered a more abstract platform where a generic event logging system would bring some value. Inspired from time & synchronization oriented artistic performances, we gained motivation to create a system which would help "use the past to shape the future", in the sense that things that happened could (selectively) re-happen, possibly extended or manipulated with an artistic twist.

Research Questions, Scope & Approach

The ways we chose to approach this problem are mostly based in exploration. First, we considered a number of real-life scenarios and metaphors where we believe that such a system would be useful and capable of producing valuable results. We created a number of simple applications to represent these scenarios and metaphors, and built the logging system around them, always keeping in mind that it has to be generic so we tried not to bind the system to the applications in ways that it would become an application-specific plugin. Our purpose was to test the flexibility of the platform (the Web), the levels of abstraction and generic representation that such a system can achieve and the degree of how can such a system stand in a distributed environment.

The two main research questions of this thesis are:

- How and in what depth can the information of user interaction be captured as a generic entity?
- What reliable way is there for the logged user interaction to be replayed and produce a consistent behavior of the system?

For the scope of this project, even though we provide insights on how could this information be used later on, we do not focus on the conclusion-making process as much as creating the platform for logging such interaction events & producing such datasets, and ultimately offering features which would re-produce the occurrence of these events in a flexible way. Furthermore, the platform of choice for this thesis is the Web, focusing on Web-based applications which could be used by one or multiple users, possibly in a real-time manner. The reasons for forming the scope as such are the following:

- The Web has evolved in ways that made it a platform for purely distributed systems.

- The feature sets of Web applications and the services they provide become richer and more popular every day, making the Web the prominent platform for most types of Internet usage.
- There has been a considerable amount of research for capture/replay techniques for stand-alone applications (mostly focused on Java & JVM-based technologies), yet there is not much research done in that respect for the ever-growing Web platform.

Thus, it is reasonable to say that the scope we define covers some cutting-edge, real-world systems with a highly distributed nature in all senses and allows us to touch upon fresh ground to research and conclude with interesting insights for the future.

Outline

Chapter 2 aims to position the topic of this project in a theoretical space by discussing the major components of the research and by elaborating on the different scenarios which were taken into account for designing and implementing the logging system.

Chapter 3 focuses on the technical aspect of the project, explaining the architecture design of the system, the technologies which were used and the reasons why those technologies were selected and examining the techniques that were implemented on each phase and use case. Additionally, it discusses issues that were raised by the distributed nature of the system & the application and ways they can be tackled.

In Chapter 4, we look upon some efficiency considerations with regards to the performance of the system in various cases. Chapter 5 provides a series of insights for creating a more generic and performant event logging system, by examining specific areas in which it can be improved.

The thesis ends with Chapter 6 where we present a list of ideas for future work regarding the use of such a system's results and datasets, along with certain features that would transform it into a generic, domain-agnostic, cutting edge tool. Furthermore, we reflect upon the contributions of this research, followed by our final conclusions.

Chapter 2: Background

In this chapter, we try to provide a theoretical background about the system that was built for this project. We explain the working principles of the major components of the logger, define the basic entities of interest and sketch the high level scenarios & criteria we decided to investigate. We also give an overview of the application(s) we built for exploring the features of the logging system, and attempt to enrich the application concepts with some metaphor(s) from real life, allowing the reader to place the event logger in a context more tangible than pure, abstract academic research.

Definition Of The Event

An event, as we decide to define it in the context of this project, is the product of any action or interaction which will provide an application with enough input to stimulate it to behave in the manner it was designed for. For this project, the event is the quintessential piece of information, as it is what we are interested in logging and reproducing while in the replay phase. Through streams of events, or *timelines* as they are also interchangeably referred to in this document, we get the picture of *"what happened"* while the application was used.

Since the event logger operates in the Web realm, we expect the events we are dealing with to belong to a -broad, yet apparent- scope. For example, we know a-priori that the typical target element of an (inter)action occurrence will most probably be a DOM element, and that the primitive types of the events themselves (as objects or entities) will follow the computational scheme & representation prototype of a Web-interpreted language (e.g. JavaScript). In a later chapter we discuss concepts and cases where the event object could be extended for different (more advanced) kinds of logging and replay, but for the sake of consistency with the design principles of the logging system, at this stage we will focus on the main characteristics of the event object which enable the logger to perform some meaningful work.

The main characteristics (or properties, since we are treating the event as an object) of an event that we are interested in are the following:

- **Type**, which indicates whether the event is the press of a keyboard button, a mouse click or movement, a native JavaScript event or any other event type that is relevant to the application and the logger.

- **Timestamp**, is the real UNIX timestamp of when the event occurred, typically measured at the source.
- **Relative Timestamp**, indicating the time difference between the occurrence of the current event and the previous one.
- **Source**, and identifier of the party that initiated the event occurrence.
- **Target**, which in turn is an object describing the (DOM) element of the application upon which the (inter)action took place. The target's properties are:
  - **ID**, the HTML id of the element (if it exists), given that it is a DOM object.
  - **Class**, the list of HTML classes of the element (if they exist), given that it is a DOM object.
  - **Tag-name**, the type of the HTML target element (div, form, etc).
  - **Text-content**, the inner content of the HTML target element, if it contains any.
  - **Type**, the value of the HTML *type* attribute of the target element, if it contains such an attribute (this attribute is usually a property of HTML form elements).
  - **Outer-HTML**, which is the source HTML code which describes the rendered target element.

Many of the event properties are optional, especially the ones which refer to the target. We want that for two main reasons: (1) the Web is a dynamic environment, and Web development is quite flexible regarding the definition of the elements of a web application, thus we want to be able to identify a target with as minimal amount of information as possible, and (2) we want the logger to treat event objects in the most generic way that it can, and that can be achieved by enabling a more loose event object structure.

System Overview

In order to provide with a high level overview of the logger, we want to list the main components which resemble it, as well as the phases in which its flow takes place.

The logging system integrates and functions in three main phases:

- **configuration**, where the nature of the application (multi/single modal), the types of events to be logged and the ways the replay is performed are defined,

- **event logging**, where the actual logging is registered and communicated from the application to the logger, and
- **event replay**, where an event stream is retrieved from the logging system to the application and the its behavior is replayed by the means of triggering the interaction events in that stream.

The aforementioned phases define a number of data flows between the application and the various components which resemble the logger. Specifically these components are:

- A set of **capture & replay mechanisms**, parts of which (typically concerning the types of events and the modality configuration of the application) are produced during the configuration phase. These mechanisms exist in a *client-side* context, a concept which is discussed extensively further on in this document.
- A **middleware layer**, which acts as a gateway of computation & logic between the application (specifically the capture & replay mechanisms) and the back-end of the logging system. Depending on the requirements and the scenario, this layer can be *fat* or *thin*, with regards to how rich functionality it has and how much computation and communication it needs to perform in the flow.
- A **API & storage server**, which exposes a set of web services to the application and/or the middleware for all relevant actions regarding storing or retrieving events & event streams, as well as maintaining states about the stage of the logging or retrieval/replay phase and generally performing global computations. Furthermore, the server maintains a datastore with all the relevant information about the interaction events and the participant(s) of the application.

Figures 1 & 2 demonstrate the architecture and data flow in the logging phase and the replay phase respectively. We will revisit each component from a technical perspective in Chapter 3.

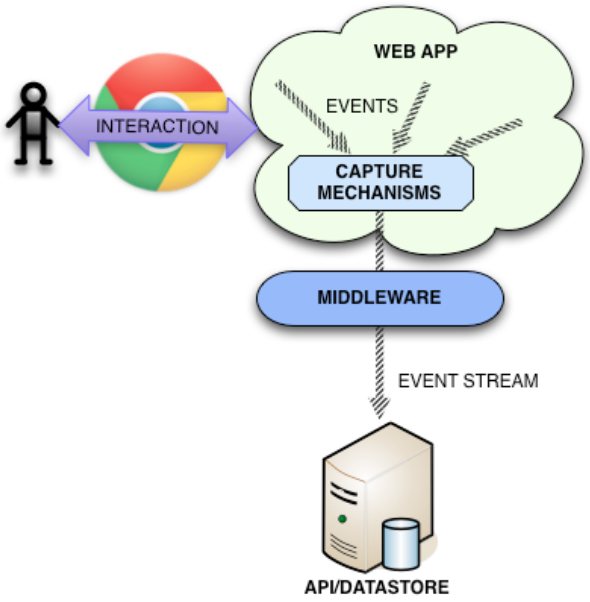

Figure 1: The flow of event logging.

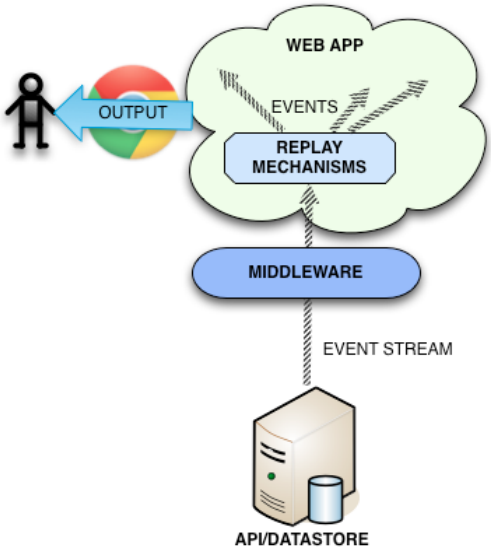

Figure 2: The flow of event replay.

Sketching Of Scenarios

In this section, we examine the cases which affect the design, implementation & feature set of the logging system as well as the application(s) under which it will operate. Each scenario is defined by applying a set of criteria of how should the application work, and in

turn it shapes the design process of the system. At this stage, we are interested in two main criteria: the application modality and how critical is the event replay operation.

With the term modality, we refer to the level of collaborative quality in the application usage: by single-modality, we imply that the application is used by one user at a time, whereas a multi-modal application is one that can be used by multiple users simultaneously (e.g a multi-player game or a voting system).

Considering the critical nature of the replay, we mainly refer to how strongly are the event occurrences coupled with each other, either in a causal or a temporal manner. This section elaborates on how each of the cases above affects the approach & risks of designing and implementing the logging & replay features of the system.

The application we built to implement & study the logger was created in versions in which we combined a number of variations between critical & uncritical replay cases of single and multi-user modes.

Single versus multi-modality

For this project, we are generally interested in applications which encourage the user to interact with the interface, because this is the main type of information we want to capture: events of (inter)action that will make the application expose its behavior. As a rough overview of our thoughts regarding application scope, our focus included applications which make the user participate in some sort of a “dialogue”, either with the application components or with other users or participants (e.g Web-based games, chat applications, voting applications, art-driven Web-based installations, etc).

In our context, a single-modal application is one where the aforementioned “dialogue” takes place between a single user and the application’s components, for example a single-player game or a drawing application. In this scenario, we have the a-priori knowledge that the timelines logged by the system originate from one source only, since there is only one user who interacts with the application. This knowledge brings specific implications when designing the logger and shapes the system in the following ways:

- When storing an event, the identity of the source is not critical because there is only one source and, therefore, only one recipient of a timeline to be replayed.
- The event logging & timeline retrieval API does not have to consider distribution of events and timelines to multiple recipients.

- The middleware layer between the application and the storage/API server needs not be aware of any peers, because there are not any.

Regarding the scenario of a multi-modal application, the fundamental difference with the single-modality case in the application level is that the input -and therefore the event streams- originate from multiple sources, often occurring simultaneously (e.g. multi-player games or collaborative art performance tools). This fundamental difference adds more depth to the design of the logger, bringing a number of intricate matters to be handled by the system.

Firstly, we know that in the multi-modal scenario the source where the events originate from matters, because the timeline can consist of events that come from different sources. The identity of the application users or participants (whether it is explicit or implicit -a concept we are discussing in detail in Chapter 3) matters and needs to be taken into account both in the phase of logging the events as well as in the phase of replaying a timeline back to the application.

The second main concern for the logger in this scenario is the temporal relationships of the event occurrences and their synchronization. Since we have multiple sources interacting simultaneously and in real-time, the logger will most probably have to process events which originate from machines which do not have fully synchronized system clocks and behave under different network schemes (which adds a variety of communication delays in the whole process), so computing the relative times of event occurrences in a timeline of such a diverse nature of origins is not a trivial task. We thus had to consider various mechanisms for maintaining a synchronized environment, also depending on how critical the replay is (more on this concept in the next section), keeping some state infrastructure on the server side and ensuring that a multi-modal timeline reliably represents the original chain of interactions in the application.

Uncritical versus critical replay

As we denoted already, the most essential feature of the event logger is its replay functionality, by which the user(s) of a given application will be allowed to fetch the stream of events they created with their usage and make them happen again in the application environment. In this feature, we wanted to provide the user(s) with certain degrees of flexibility, such as the ability to selectively replay a subset of events from a given timeline or create multiple timelines with different properties. To achieve such flexibility, we used techniques which applied both in logging-time and in the selection phase of the event stream, and we

discuss their technical details in a later chapter of this document. Similarly to the modality factor we discussed before, in this case we had to take the replay criticality into account and design our replay techniques based on it. In order to decide on our approach, the basic question we had to answer for each scenario was: *How important is a **correct** behavior of the application while/after replaying a timeline?*

At this point, it might be necessary to put the term ***correct** behavior* into context. When a user interacts with an application, they obviously expect to get some results back. The nature of these results is vastly variable; they can be computationally generated, multi-media output, a networked invocation or an interaction with another application or user, just to name a few examples. For the scope of our project, we see correctness as the combination of the application-specific set of results and the expectations of the user as to what is the minimum information they need to get back from the application. For example, in a color-changing application, if the user commands the application to change the color from grey to green and all they want to see is a color changing, without caring about whether the changed color is green, then in the case that the application changes the color to red is still considered to be a correct behavior (even though probably something is going wrong in the application internals), from the user's aspect. This approach might seem unconventional, but if we consider our initial motivations and the fact that one of our targets for this project is the possible artistic expression through real-time computer systems (in which case randomness is a usual means for sparkling artistic glory), it becomes valid because it widens the horizons for interpreting the timelines as well as implementing the logging and replay functionalities of the system. Of course, if the application architecture dictates that there should be a deterministic relationship between usage and result output for its function to be sensible & correct, then the definition of correctness takes its traditional deterministic form and the user's expectations should not differ from the application-set paradigm of correct behavior.

We use the above note about correct application behavior to speak about the criticality of the replay functionality. In our system, we look at the importance of a correct application behavior when replaying the events that happened during usage. And in the scenario where the replay is not critical, we imply that the system behavior (as it is formed by the replayed timeline) can be acceptable even if it does not match the event stream from the original usage in a one-to-one basis. Such a concept refers to applications where there are no causal dependencies between interaction events, regardless of the application modality and multi-user support. Furthermore, uncritical replay applies to systems where there are no temporal dependencies in their usage, and if there are they do not matter when replay-

ing a timeline. Thus, in the case of uncritical replay we most likely speak about applications through which the user(s) aim to produce non-deterministic results, and possibly some artistic expression through replaying their interactions.

The most profound implication brought by uncritical replay when designing the logging system is the fact that we do not have to perform extensive "event housekeeping", as we do not care much about the ordering of the events in time, the synchronization of interaction steps or sometimes even the source of the events, in the multi-modal case. Thus, the logging and replay phases are fairly simple and they require minimal specification. Additionally, the event structure as an entity becomes somewhat loose, in the sense that it needs to contain less meta-properties as there is a limited amount of information that the logger needs to know about the event objects in order to perform the logging & replay functions.

The landscape changes dramatically -as it is expected- in the case where replay has a critical nature. By critical replay we mean that the replayed event stream should reflect the exact original timeline which was produced by usage, or it should at least follow a set of strict criteria to achieve the desired (and inherently correct) behavior of the application during replay.

Typically, critical replay is required by applications which produce interaction timelines with events that are causally -and thus also temporally, from a timeline perspective- dependent on each other. It is important for such applications that the relationship of *"event B happened because event A happened"* is maintained while replaying a timeline, in order for the application to behave correctly, and this means that the logging system has to keep the correct ordering of the event occurrences into account. Additionally, since from a timeline perspective (and for the scope of this project) we want to assume that causally dependent events are also temporally dependent (rephrasing the causal concept to *"event A happened first and event B happened second"* indicates that events A & B are tied in time), the logging system needs to carefully handle the time of each event occurrence and order them accordingly in the timeline to be passed to the replay phase. With all the above in mind, and especially if we consider the scenario of a multi-user web-application with critical replay required, it is quite clear that achieving consistent logging and replay of events which occur in a distributed context and emerge from multiple sources with variable network and otherwise existent overhead is not a trivial issue. Chapter 3 discusses how we decided to tackle these issues in detail.

Application & Metaphor: A Simple Voting System

The application we built to test, prove & improve the features of the logging system had to comply with a set of criteria:

- It had to be a Web-based application.
- It should allow a single user to use it, as well as multiple users simultaneously.
- It needed to behave in a real-time fashion, with regards to the output the user(s) would receive as a product of their interaction.
- It had to represent a real-life metaphor, to assist our research and further argumentation regarding the usefulness of the logger.

For the purposes of this thesis, and given the fact that the application is merely a "playground" we used to base our case studies on and not the objective of the project, we wanted to create an environment that is minimal in terms of its visual interface & feature set, yet functional enough so we can observe the behavior of the logging system, as well as a representative example of a hypothetical real-life use case scenario.

The application we decided to build therefore consists of a fairly simple interface and functionality, and it supposedly represents a primitive, color based voting system, with which the user can express their stance on a given subject in three different levels, and each level is represented by a color that gets registered after their voting action and is shown as output on a square area, as Figure 3 illustrates. The user is able to act on the controls of the application indefinitely, and each time the color they choose is painted on the square area in real time.

The application was built in two versions, one where only one user can vote at a time and their results only appear to them, and one where there is collaboration involved and multiple users can vote simultaneously, and each vote appears in all participants' output square areas.

The way with which the user can interact in this application is by clicking one of the color labeled buttons above the square area. Thus, the event logging configuration we applied was tracking the click events on these buttons, together with the complete context of interest and event-specific properties we discussed in an earlier section of this chapter. We provide with a detailed technical description over what technologies we used as building

blocks to assemble the application's architecture, communication scheme & interaction with the logger in Chapter 3.

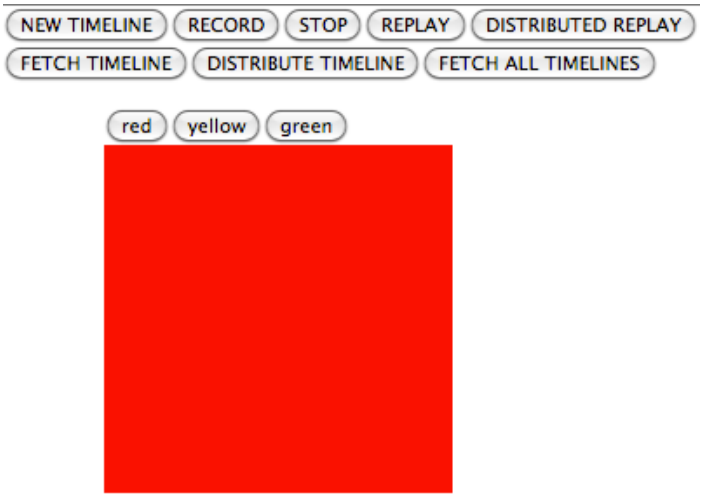

Figure 3: The interface of the application we built to test the logging system. The controls at the top of the window are application-independent and are used to manage the logger's features.

Chapter 4: Performance & Efficiency Considerations

With the analysis of the previous chapter, we solidly established the fact that the logging system itself as well as the applications it aims to act upon belong to the group of the so-called distributed systems. And in such systems, characteristics such as scale, transmission rates and spatial distribution are most likely to manifest anomalies of various types, affecting performance & efficiency and leading systems to deviate from their design and expected behavior.

In this chapter, we will examine the expectations of how the logger should perform under a series of real-life scenarios with properties which are likely to pose threats on the system's performance, and try to identify the possible points of failure within its architectural components. We will focus on two main aspects that can be proven to be soft-spots of the system: scalability, both in terms of work load & input density as well as geographical distribution, and synchronization, which is an important metric regarding the efficiency of the logger, given the fact that we want to log, process & replay real-time events.

Scenario: High Event Density

Let us consider the following scenarios:

- The participants of a large-scale live concert use the collaborative voting system described in the previous chapter to express how appealing the performance of the artist(s) is to them, or how adequate is the quality of the audio-visual experience at the place where they sit/stand, all in real time. The information generated by each user's interaction is processed by a (possibly) centralized service, followed by a multi-cast type of propagation of this information to the other participants, perhaps based on a "focus-group" paradigm (e.g. participants share their impressions on sound quality with the ones located at the same general area as themselves).
- A number of people participate in a collaborative, multi-player Web-based game which is operated by the means of an advanced motion-capture controller device, such as Microsoft's Kinect sensor.

From the event logger perspective, both the aforementioned scenarios have one main characteristic in common, and that is the vast amounts of events being generated by the participants over relatively small time intervals. For the sake of context, it is appropriate to indicate that the Kinect sensor captures depth and color images simultaneously at a frame rate of up to 30 fps and the integration of depth and color data results in a colored point

cloud that contains about 300,000 points in every frame. And if the logging system operates with a high event resolution, one can easily notice the increased work load for the system in terms of datastore operations (database writes & reads). Furthermore, in both the above scenarios we have some pieces of sensitive information carried along with each event, with the identity of the source (or the source group) being the most prominent. Such event properties bring additional complexity during the event processing by the logger and, with the event density being so high, the processing overhead could increase exponentially.

In front of such landscape, the logging system in challenged to be capable to perform well with increasing work load, or achieve work-load scalability, and thus be able to easily expand and contract its resource pool to accommodate heavier or lighter loads or number of inputs.

The impact of this scenario would primarily become most visible in the back-end of the logger's architecture, bringing immense load in the back-end server and especially database, where all the generated events need to be stored and, later on, retrieved. With the current configuration of the system's components, this block in the architecture needs to be thoroughly protected because it currently suggests a single point of failure, which is a hazardous property in any distributed system. The first (and the relatively simplest) measure to be taken for this scenario is to scale the back-end server in a vertical manner, that is to say to add resources to the node, typically involving the addition of CPUs or memory to it. This technique would increase the capabilities of the back-end & storage server for handling bigger loads in terms of processing, storage and memory allocation for the dynamic data-stores. The technique of horizontal scaling could be applied as well in this case, meaning that the processing & storing phases could be separated and being performed from two different servers, instead of having only one machine doing both operations. However, even in this case the database server should be a sufficiently powerful machine, especially with regards to memory size and disk space, if it has to store such huge amounts of data.

Furthermore, the table & document structure that is currently defined for event storage in our MongoDB instance would have to be modified in order to sustain vast amounts of data. Since the structure currently dictates that each user has their own document in the table, and all the events they produce are appended in this document (the MySQL equivalent would be similar to a row in a table in which columns are dynamically added), runs the risk of quickly exceed MongoDB's document size limit, which currently is equal to 16MB.

With such vast event volumes, one can safely assume that this limit will indeed be surpassed and from that point on, actions need to be taken from the programmer's perspective. Two approaches are identified:

- introduce an algorithm to wrap the document into a new one once the size limit is exceeded, or
- re-structure the table structure so that users and event streams are connected and related with each other.

The first alternative follows a simple approach which trusts the existing structure, however it has the long-term disadvantage of adding extra computation effort in the server for wrapping the documents, while maintaining consistency around the table and gathering them back again when they are being retrieved. On the other hand, the second technique requires more mental effort for coming up with a solution which re-defines the correlations within the database and takes advantage of the efficient and performant tools of the NoSQL system (which requires taking distance from the traditional way of thinking in the world of relational databases). However, albeit a fairly intensive mental task, our expectation is that such a technique would benefit the capabilities of the database when handling vast loads of data, both upon storage and retrieval. Regarding data consistency, one must in this case trust on the eventually consistent nature of NoSQL, meaning that transactions will not be present (which is for the best in this scenario, since the extra computation transactions need would be prohibitive on such data loads).

From the communication & synchronization aspect, given that we want to protect the (current) single point of failure of the architecture, there are a few techniques which can be applied, namely the middleware of the logger can be extended in such ways that it relieves some of the load from the back-end. More specifically, since each middleware node is dedicated to a small number of clients (if not only one), a layer of caching can be implemented on such a node, using some smart caching algorithm to store the generated events in a fast and light-weight, key-value based database system such as Redis and periodically synchronizing the cached events with the back-end server. The periodical communication between the caching modules of the middleware and the back-end server could be done either by having the back-end poll the middleware in a manner of status maintenance, or it could happen in the opposite manner, with the middleware nodes pushing the data to the server either via a RESTful API call or via WebSockets. This way, storing events in the central datastore happens in a less rapid rate and retrieval may not even require communication with the back-end server.

Regarding temporal consistency between the clients and the events they generate, the middleware can also contribute on this front. In co-ordination with the back-end server, an instance of the Berkeley synchronization algorithm would notify the middleware nodes about the time in the server, thus creating a state of time offset on each node. This way, every middleware node would be synchronized with the back-end server's master clock, and they would calculate the relative timestamps of each event based on their respective offset which will be derived from that master clock.

Scenario: Geographically Distributed Interaction

In the previous section, we reflected upon the logging system's scalability & synchronization consistency potential against scenarios which had the specific characteristic of high density of events coming in the logger as input, creating an increased work load along the complete architecture stack. In this section, we will examine how the logger would accommodate and maintain the same qualities in cases of a different nature. We will look at scenarios which introduce performance & efficiency bottlenecks caused by common characteristics of real-life networking: the wide geographical diaspora of nodes & clients which immediately implies differences in transmission rates, network delays & possible transmission failures.

Let us consider the following scenarios:

- A real-time, Web-based, pan-European polling system, in which citizens of the whole continent are called to place their stance on certain matters concerning the European Community.
- A collaborative music synthesis Web-application, where users are located around the world and each of them controls virtual instrument, having their contribution add up to a general musical piece which is being synthesized in an ad-hoc manner.
- An augmented reality multi-player Web game, where each player takes a turn and "shows" a landmark with a cinematographic identity from the place they are currently located in the world, and the rest of the players try to guess on-the-fly what its identity is.

The common characteristic of the scenarios above is that the communication range of the application escapes the local-area network scheme and moves to a wider, possibly world-wide distribution of clients & nodes. Such wide geographic distribution introduces the most common bottleneck for any system that operates in a distributed manner, since the network heterogeneity with regards to network infrastructure (bandwidth, latencies etc) along

the participants imposes performance & efficiency threats. In that respect, the logging system by design has a significant advantage and an equally significant disadvantage, which needs to be addressed if we want the logger to scale reliably from geographical aspect.

The advantage of the logging system resides on the asynchronous communication scheme it uses. It is known that one of the main reasons why it is currently hard to scale existing distributed systems that were designed for local-area networks is that they are based on synchronous communication. In this form of communication, a party requesting service, blocks until a reply is sent back. This approach generally works fine in LANs where communication between two machines is generally at worst a few hundred microseconds. However, in a wide-area system, one needs to take into account that interprocess communication may be quite a few orders of magnitude slower. The logging system was designed to use JavaScript along its complete stack, thus following the natural asynchronous operational schemes of the language. From I/O operations to network communication, there is no place where some layer of the system would block its execution and wait, unless a part of some communication algorithm would strictly dictate such behavior. Therefore, extending the logging system to operate under a wide-area application (similar to the multi-modality scenario we talked about in the previous chapter) would not threat its performance in the synchronous/asynchronous communication respect.

However, there is one “flaw” in the current design of the logging system which would prevent it to efficiently scale geographically, and that is the centralized fashion in which the back-end & storage server operates, which resembles a scenery of both centralized services and centralized data. Typically, geographical scalability is strongly related to the problems of centralized solutions that hinder size scalability. If we have a system with many centralized components, it is clear that geographical scalability will be limited due to the performance and reliability problems resulting from wide-area communication. It is obviously imperative that such issues need to be tackled, and hereby we place some thoughts on possible solutions for addressing them.

Distribution of services

A centralized server that needs to serve requests at a large geographic scale, quite easily becomes the bottleneck when the requests increase and originate from a wide range of locations. And for the logging system, although it does not block communication with the asynchronous scheme it uses, the differences in network configuration of each client may cause possible anomalies in event processing.

The first -and almost inevitable- measure to address such an issue is to scale the back-end layer horizontally, by increasing the number of servers and expose the event API in a more “localized” fashion, allowing clients & middleware to communicate with the back-end that is located closer to them, thus decreasing the possibility of bad performance due to high network latencies. Additionally, and given that the applications we log are collaborative, the API servers could function as a type of peer-to-peer community, using slick communication & intra-synchronization techniques such as gossiping algorithms for spreading the system state information or voting algorithms for load management. The above technique could also be applied in a second degree of depth, by extending the middleware to carry specific parts of the API and take even bigger advantage of the geographic locality they possess in the general architecture.

The goals of this approach essentially aim at making the back-end system more flexible, distributing the load along a larger number of servers while maintaining consistency regarding the incoming event operation requests & ultimately achieving communication transparency on the client-side while logging as well as replaying events.

Distribution of data

As far as creating a possibly failure-prone distributed system, centralized data is essentially as bad as centralized services, for the obvious reason that all data-related requests (which for the logging system are approximately 95% of the total request volume) end up in one place, creating great work load and over-saturating the connection links. Being a visible threat to the capability of the logger to scale geographically, it is an issue that needs to be tackled as well.

In a similar fashion with the distribution of services, the first step towards addressing the issue would be to scale out and add more machines at the data storage layer. The immediate implication for the logger’s architecture is that now the back-end part (API and event processing) is physically separated from the data-storage part, which as a side effect relieves work load by having the back-end server(s) handling & processing the event requests & responses, and the database server(s) taking care of the storage, retrieval & data manipulation operations.

With multiple database servers available, there are a number of techniques one can use to improve performance & geographic scalability. The most common practice in this case (and the one that is proven to work with good results) is replication, which defines a master/slave relationship between the database servers with the master(s) being the one(s)

responsible to keep the slave(s) up-to-date with the data that have been inserted/updated/deleted from the database. Replication is a very smart technique and can very much improve the efficiency and performance of the logging system when operating under an application of a large geographical scale, with the possible overhead of having to ensure data consistency, namely that all the replicas will as soon as possible receive the latest changes in the data, regardless of where they happened. With our design decision for a database being MongoDB, a NoSQL system without transactions (the most common tool for orchestrating consistency algorithms), we will have to trust the so-called eventually-consistent model it works with, meaning that the data will be consistent at some point in time.

Moreover, given that currently we have read & write operations in the logger being finely independent (event logging only involves writing in the database, and event replay only involves reading from the database), one could define groups among the database servers, with one group receiving & performing all the write operations (probably the masters in the replication scheme) and a second group handling all the read operations (probably the slaves in the replication scheme). Of course, such groups would need to be defined with certain criteria in mind, most important of which is the locality requirement, which dictates that the distributed back-end servers we described in the previous section should be neighboring with database groups which contain both read-only & write-only, thus providing the clients with access of the complete feature set (event logging & event replay) from their neighborhood. Such technique is expected to actually assist on maintaining data consistency by introducing a strongly recognizable group of replication master servers, which could communicate in a fashion similar as the one we suggest for the back-end servers in the previous section as well as keeping their neighboring slave replicas updated. It does not bring any serious additional overhead to the overall replication mechanisms (the data need to be propagated to all the replicas anyway), instead we expect that it decreases the distribution of load at one more order of magnitude, while in the meantime it takes advantage of the locality created by this grouping.

Figure 6 summarizes in a schematic manner what we discussed in this and the previous section.

Synchronization

For every system that operates in a wide geographical scale, managing time can -and, most likely, will- be a non-trivial problem. And it has been made quite clear in this thesis that for the logging system, working with real-time events, timing & synchronization plays a

crucial role. In all the application scenarios we mention in this section, the users can be located anywhere in the world, which implies that their systems will not be synchronized and their clocks will probably be set to a different timezone, with both these facts being reflected to the events they produce. Thus, the aim is for the logging system to store & retrieve the event streams in the right order by using a unique & reliable metric to determine their temporal relationships and calculate their relative timestamps.

Having set the scenery, our first intuition is that it would be too much overhead to force synchronization exclusively on the client-side, which instead needs to be achieved between the middleware & back-end layers, with the clients making as minimal an effort as possible.

Starting from the back-end servers, the last layer before forwarding the events for storage, absolute synchronization is crucial. The servers resembling this group could be synchronized with each other by using the Berkeley algorithm, with an elected server being the master (and getting synchronized by a universal clock via the NTP protocol) and the rest of the servers being the slaves and getting synchronized with the clock of the master.

After ensuring synchronization in the back-end, the servers would communicate their time (which will hopefully be the same along the back-end layer) with the middleware, and the middleware machines would then maintain the time offset based on their local clocks. Moving up to the client-side, a snapshot-based algorithm could be used in the sense of having a timer counting while events are being produced and communicating that counted time with each logged event to the middleware. At this stage, the middleware will have enough information from both the client-side and the server-side to compute the correct relative timestamp for each event, always based on the time offset of the back-end.

During replay, the relative timestamps of the event stream (and therefore the order & temporal relationships of the events) are already set, therefore the only synchronization aspect to be taken into account is synchronizing the replay of the distributed timeline at the client-side. For example, in the music synthesis scenario we mention above, if a timeline starts with the piano user playing first and the drums user coming second and we know that the drums user has a better network connection than the pianist, we still want the piano to come first during replay regardless the fact that the drummer would get their events faster. This can be achieved by having the middleware exchange a series of control messages between their connected clients and each other in a fashion similar to the TCP handshaking mechanisms, thus ensuring that all sides are ready to start replaying.

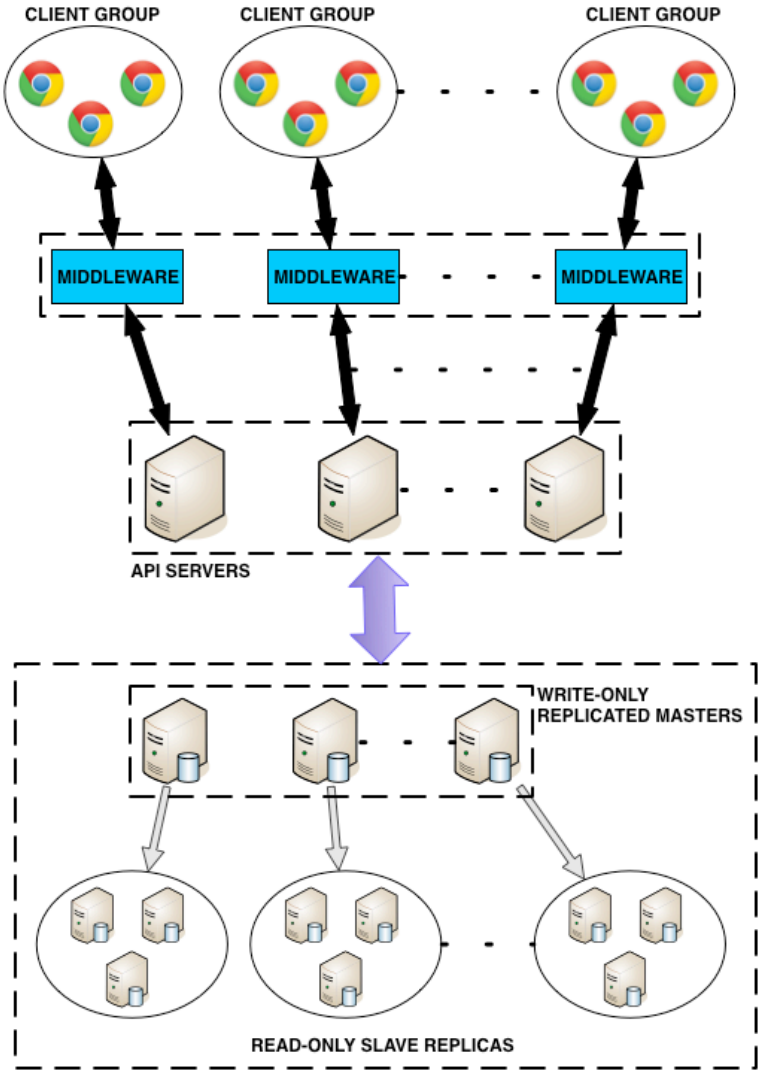

Figure 6: An abstract schematic representation of the architecture of the logger with a rich middleware layer, distributed back-end servers and a layer of replicated & grouped database servers.

Chapter 5: Towards A Fully Generic Event Logger

In this chapter, we want to explore concepts which would take the logging system from the prototype-like form it is now and transform it to a more generic & reliable system, while improving its scalability, efficiency & performance capabilities where possible.

In principle, we expect the logger to evolve towards the point of being fully generic by accommodating three main qualities:

- **Application agnostic**: the logger should be configured support different types of applications regardless their modality or design.
- **Event-type agnostic**: the logger must be able to log, process & replay any type of events the application usage could generate.
- **Context aware**: the logging system should be able to log events that are not necessarily created by direct application usage but occur as side-effects in the context of the usage itself.

Of course, the above points may resemble a somewhat utopian expectation set, however we believe that there are techniques & tools one can experiment with and hopefully move towards one or more of the directions we present here. We will elaborate on such techniques & tools for the remainder of this chapter.

Extending The Event Format For Complex Event Structures

It is a widely acceptable fact that interaction with computer platforms (including the Web) has evolved dramatically during the past decade, adding multiple and much more exciting ways that allow a user to interact with a computer, apart from the classic keyboard & mouse paradigm. The implication of this statement is that with new controller devices, new event types are defined in a programming level (e.g. the skeleton movement events generated my a Microsoft Kinect device). Furthermore, with computer software becoming more complex and programming methods such as concurrency & parallel execution becoming more widely used, one can expect that parallel or non-deterministic subsequent event occurrences become more visible within applications' execution cycles. Therefore, a logging systems that aims to be generic needs to catch up with such concepts and become able to incorporate and adapt with more complex event structures and streams.

The logging system we built for this project, considering the Web platform on which it operates, indeed has such a potential. This potential mainly resides in the fact that it is built

with JavaScript which provides sufficient reflection capabilities to log client-side non-determinism in standard JavaScript running on unmodified browsers. With this insight in mind, the logger could be extended to capture "behind-the-scenes" activity such as the firing of timer callbacks and random number generation, or even network events such as asynchronous AJAX calls or WebSocket communication. Such capabilities of the language give additional potential to the logger, that of being aware of the application context which may react to an internal or external event occurrence. We look into depth on context awareness in the following sections of this chapter.

Introducing Reactive Programming Principles

Functional Reactive Programming, or FRP, is a general framework for programming hybrid systems in a high-level, declarative manner. The key ideas in FRP are its notions of *behaviors* & *events*. Behaviors are time-varying, reactive values, while events are time-ordered sequences of discrete-time event occurrences.

One can already notice the particularly intuitive matching of terms & keywords between FRP and the logging system: events & behaviors is what interests us in the context of this thesis, and in FRP the same notions exist and, although they are defined in a lower & more conceptual level, it seems to be a technique which would integrate in a natural way with what we tried to do in this project.

From a programming standpoint, there are two main matters FRP can appear useful with. First is the JavaScript callback issue on the client-side event logging mechanisms. It is well known that the asynchronous & event-driven way that JavaScript operates is very much based on the primitive construct of callback functions. For example, when a browser completely loads a document, a user clicks a button, or an Ajax request is fulfilled, an event triggers a callback. JavaScript's functional nature makes it extremely easy to create anonymous function objects that you can register as event handlers. Naturally, this way of handling events is the one we used for this project as well. However, if one strives for making the logger capable of generically logging any type of events from the applications, one must make a considerable amount of effort composing callback functions for at least any type of event group the logger might encounter, if not (in the worst case) just any type of event object that could be produced by a Web-application. This approach, although it could possibly work, makes for long code which hard to comprehend & maintain and ultimately it may even harm the generic nature of the event logging mechanism by essentially turning it to a tailored-down solution for the application. FRP tackles this issue by focuss-

ing more on the behavior of the target of the event occurrence, and while monitoring this behavior it internally triggers handlers -which are regular, programmer-defined JavaScript functions which are internally treated as callbacks- to process the stimuli that changed the behavior of the target element, namely the event itself. And since the event is a core entity of FRP and it is therefore defined with excellent detail and flexibility, such technique would allow for logging more complex (possibly indirect) events in different rates and configurations.

The second, and maybe more important, matter where FRP would be proven helpful is the context awareness of the logging system. In a serious gaming environment, we are very much interested to record, replay & reflect upon not only the direct event which caused some behavior in the application, but any side effects of that event to the rest of the application, its contextual landscape. In fact, it is very common that the context which becomes affected by a certain event occurrence is much more useful and insightful for many types of analyses. With this unique FRP feature of monitoring the behavior of an element that is a possible event target, one could enforce context awareness at configuration time, setting the logging mechanisms to watch for changes in the behavior of certain elements that are known to react in a side-effect manner when some event occurs somewhere else in the application environment. This approach would allow for very interesting post-hoc analyses with the purpose of identifying possibly complex causality patterns between event occurrences.

Smart Loggers

For the purposes of this project, the logging system was configured and built to act “stupid”, in the sense that there was no intelligence whatsoever during the record or the replay phase, with regards to conceptual manipulation of events. Essentially, the logger would just record the complete volume of events for a defined timeline, and it would eventually replay the same amount of events it recorded. Although this approach is effective for an initial stage, the logging system would definitely improve in many aspects (including the logger’s capability of being even more generic) if some degree of sophistication could be applied on the configuration phase of the logger.

An initial step towards making the logger a bit “smarter” would be the support of definition of rulesets during configuration, allowing for creation of implicit or explicit rules that could modify certain aspects of event logging & event replay, based on user-defined criteria. For example, if we consider the voting application we built as a testing environment for the

logger, one would want to record discreet alternating votes by all the users, meaning that the logger should ignore every event of a user after their first one, until another user generates their event. Such a requirement is absolutely valid if one wants to identify causal patterns between users with a bit smaller granularity or less noise in the events. Also, this example can be generalized as a configuration parameter which could define the resolution of events that are being logged, and it would potentially benefit the performance of the logger by decreasing the work load.

Another configuration feature which would be very useful is the support for declaring how critical time is when recording & replaying events. Coming back to the voting system example, it could be that one is interested only on the sequential patterns of event occurrences and not their exact timing information & temporal relationships. This feature would significantly improve the processing of the event objects in the middleware and the back-end servers in the logging phase, as well as the efficiency of the replay mechanisms.

Finally, and in combination with the FRP principles we described in the previous section, it would dramatically improve the depth of event logging if the logger would support defining a context within the application during the configuration phase. Allowing the user define the a set of elements or stimuli they are interested in makes for a purely generic method of creating a context-aware tool.