Elmsvuur: a Multi-tier Version of Elm and its Time-Traveling Debugger

Jeff Horemans, Bob Reynders, Dominique Devriese, and Frank Piessens

imec - Ditrinet, KU Leuven
`first.lastname@cs.kuleuven.be`

Abstract. Debugging dynamic web applications is challenging in many ways. Applications intrinsically consist of (at least) a server and a client program, each deployed on different nodes. Because of that, web applications behave like distributed systems and debugging them faces much of the same problems like managing state and heterogeneity of nodes. For web applications the latter problem also applies for the distinction between client and server code. These sections are typically written in different languages which further complicates the debugging of a web application.

The most common solution is dealing with each program layer individually using language-specific debugging tools, but this does not give an overview of the entire application flow.

Multi-tier languages allow programming a web application in a single language as a single application. They are primarily designed to offer advantages with regards to developing web applications, but this opens up new possibilities for debugging as well. We propose Elmsvuur, a multi-tier version of Elm and a time-travel debugger for it. The debugger operates as a single tool for the whole client/server application. We discuss how advanced timeline debugging features, such as resuming from different points in the history, allows programmers to find bugs across tiers.

1 Introduction

Debugging dynamic web applications is challenging in many ways. Applications consist of (at least) a server and a client program, each deployed on different nodes with client and server often written in different languages. This variety in programming languages further complicates the debugging of a web application and makes it difficult to maintain a global view of the application when finding bugs.

Multi-tier Languages. The classical process of developing a web application consists of writing programs for the different programming layers or “tiers” of the application as well as (parts of) the logic for communication that needs to happen between them. With JavaScript being the most widely used language for client applications and various options for the other layers, they are mostly written in different languages. This may also require writing code that handles sending

and receiving information between programming layers, as well as converting that information to a format that both parties can understand.

Multi-tier languages were created to solve some of these problems. They provide a developer with one language to create web applications, without the need to write programs for each layer in different languages. Instead, the code for the programming layers underneath is generated by the multi-tier framework based on the multi-tier application source code. Links [1] and Hop [2] were among the first multi-tier languages to be created, with Ur/Web [3] and Hop.js [4] as more recent examples. Having only one source language means that the programming process is less partitioned, and depending on the design of the multi-tier language they can also offer various abstractions, most notably in terms of communication.

For debugging, similar simplifications apply. Multi-tier languages make it possible to have a single debugging tool for the whole application, in contrast to debuggers for each programming layer. Such tools, like the Hop debugger [5], also have the benefit of having access to the complete execution of the application.

Time Travel Debugging. Finding and resolving bugs often comes down to re-tracing a series of events that led to the bug. This can be quite repetitive work, especially if testing many different scenarios with the same event history. A technique that can aid in such situations is time travel debugging ([6, 7]). This style of debugging makes it possible to travel through the history of the application and inspect it at previous points in the execution time line directly. Other features can be: pausing the application, replaying past events (possibly with hot-swapped code), continuing from a previous point, and more.

To make this all possible, time-travel debuggers need to store some kind of information about the state of the program for each point in the timeline and about the event that caused it. Not only to be examined afterwards, but also to potentially rerun events.

Multi-tier Time Travel Debugging. We propose to combine the technique of time travel debugging with a multi-tier debugger for web applications. This combination results in a debugger that has more insight in the execution of the application because of its multi-tier nature and has time travel capabilities that apply on a multi-tier level. To accomplish this though, there are some challenges to consider.

- Multi-tier languages offer one language to develop applications, but once deployed, they consist of different programs for the server and each client like any other web application, with each of those programs having their own timeline of events. So inherently a multi-tier time travel debugger needs to deal with multiple timelines and their interaction, distributed over client and server. Figure 1 shows these different timelines with every node representing an event. It also shows how the global timeline of the application is composed out of those client and server timelines and how some of their events can be interconnected. In that context, it is important to consider how the

Fig. 1. Concept of multiple timelines

debugger constructs that global timeline and how time travel functionalities are implemented with that architecture.

- Altering the global timeline of the application means potentially altering each of the individual timelines and their state. This process requires that after the change, the states of the server and clients end up in a consistent state.
- When the application is paused, the debugger needs to deal with events that can still be received from pending client-server communication or finishing side-effects.
- After changing the timeline of the application, events can be received that originated from a previous timeline and have no place in the new one. So the debugger has to be able to discard those accordingly.

Multi-tier Time Travel Debugging: an Implementation for Elm. Certain program architectures lend themselves better for certain styles of debugging than others, the design of a programming language or framework can largely affect how easy a debugger is constructed and used. In this paper we focus on a multi-tier debugger specific to Elmsvuur¹, our own multi-tier framework based on the Elm language [8] that lends itself well to event inspection. In fact, we point out that our design

¹ Elmsvuur is the dutch name for St. Elmo's fire: a weather phenomenon that produces electric discharges from sharp objects (like ship masts) in certain atmospheric conditions. The discharges remind us of messages escaping a single tier's boundary.

and implementation are rather specific to Elmsvuur and its architecture and the approach of multi-tier time-travel debugging would probably be considerably harder to implement for other multi-tier languages. Our debugger has the same design goals as the time travel debugger in Elm². It is easy to inspect incoming and outgoing events and effect these have on the state of the application but it is not within the scope of the Elmsvuur debugger to inspect the pure computation that calculates such intermediate states.

Elm is a pure functional language that compiles to JavaScript, meant for developing client-side web applications. It distinguishes itself from others by having the main goal of offering a language that makes developing web applications easy and intuitive, instead of trying to solve shortcomings of JavaScript. That vision fits closely with what multi-tier languages try to achieve for developing web applications and with many users experimenting successfully with Elm on the server using Node.js, it is a promising candidate for becoming a multi-tier language.

In terms of debugging, Elm already comes with a time travel debugger which has recently been integrated in its main development tool, Elm Reactor.

Elm applications are written by using the Elm architecture, a simple pattern for developing applications. Elm enforces at least the main module to have a separate definition of the program's state, user interface and event processing logic. Elmsvuur extends this pattern to a multi-tier level. Its debugger, written as a multi-tier library, wraps itself around an Elmsvuur application to add its instrumentation.

Overview. To summarize, the contributions of this paper are:

- A proposal for applying time-travel debugging in a multi-tier language, taking into account issues like separate timelines, causal dependencies across timelines and the interplay between events and timeline modifications.
- The design and implementation of Elmsvuur, a multi-tier version of Elm, built around a multi-tier version of the Elm architecture.
- A complete prototype implementation of both Elmsvuur and its debugger, demonstrating their practicality³.

The remainder of this paper consists first of a look at the architecture of Elm and Elmsvuur in Section 2. Then we give an overview of the Elmsvuur debugger and its functionality in Section 3. Next, Section 4 offers a closer look at the architecture and design of the debugger. Next, we discuss future work in Section 5 followed by a final rundown of our contributions in Section 6 and finally conclude with related work in Section 7.

2 Multi-tier Elm

Elmsvuur is a natural extension of the Elm architecture to a client/server setting. Let us first look at the Elm architecture and then its multi-tier variant.

² <http://debug.elm-lang.org/>

³ <https://github.com/JeffHoremans/elm-multitier-examples>

2.1 The Elm architecture

Fig. 2. The Elm architecture

Basic. Elm requires that every application is structured according to the Elm architecture, shown schematically in Figure 2. In its most basic form, there are three core elements: the model, the update function and the view function. The *model* is a data structure that stores the current state of the application. The *update* function is responsible for converting the current state (or model) of the application into a new one, based on an incoming message (**Msg**). Finally, the *view* function defines how to turn the current model into a structure representing an HTML tree. Such a structure defines what the user interface looks like as well as which input elements trigger new messages for the Elm program. The only responsibility of the programmer is to describe how to convert incoming data to outgoing data using these three elements. It is then up to the Elm runtime to connect those elements and perform actions accordingly.

Side-effects. The basic architecture does not allow general communication and reaction to the outside world. For that, two elements are added to the basic architecture: subscriptions (**Sub**) and commands (**Cmd**). Just like the HTML construct the view generates, both are data structures that describe something for the Elm runtime to perform and both result in messages that return to the program. Subscriptions describe what Elm needs to listen to, for example a web socket or the current mouse coordinates. Commands on the other hand, describe something Elm needs to do, for example perform a HTTP request or print to the console. Effects in Elm are thus represented as data. They are not performed immediately, instead they are passed to the runtime which actually performs them while returning their results as messages for the update function.

Extensibility. The *model*, *update* and *view* structure is imposed on an Elm program by the main value, which is not an imperative, but a declarative main that expects a *model*, an *update* function and a *view* function. When one of those elements, or the entire main module becomes too complex, these declarations can

require `MultitierCmd`. These new commands, can be one of two things, a regular Elm command or a remote procedure call command which we discuss later.

To summarize: the client tier is defined as a client model, a client update function and a client view function.

Server Tier. Up to this point, the resulting server program only contains default framework code to run and host the client program. To be able to extend the server application code, the Elmsvuur architecture defines server variants for each of the Elm architecture elements. The server defines its own model and update function and allows for server-side commands and subscriptions.

With the full Elm architecture on the server it would be possible to listen to the server's clock with subscriptions or write files using commands.

Asynchronous Remote Procedure Calls. The final element of the Elmsvuur architecture is the use of remote procedure calls which allow clients to perform actions on the server. These model regular HTTP requests in a multi-tier fashion and fit naturally in the asynchronous Elm architecture. They are designed as data structures that describes RPC actions and are interpreted by the Elm runtime. RPC commands may update the server model and can even result in other server commands or subscriptions. On completion, the result of the RPC is piped back to the client update cycle in the form of a message.

As mentioned above, RPCs are a type of multi-tier command but unlike regular client commands they describe something that needs to be done on the server instead of the client.

3 Multi-tier Time Travel Debugging by Example

Fig. 4. Screenshot of our room reservation application.

This section presents the Elmsvuur and its debugger using an example room reservation application. This application allows users to schedule a conference room under a given reservation name by clicking available slots. Figure 4 offers

a screenshot of the application. The list of rooms with their reservations are maintained on the server, with clients performing calls for updating reservations and the server notifying clients with changes. For simplicity our interface only shows a single room and a single reservation slot. We strongly encourage the reader to explore the example and the debugger. The entire scenario as it is described here can be tested on our project⁴.

To help illustrate the debugger we consider a simple scenario. Two clients try to schedule the same room and both succeed in doing so, thus exposing a bug. Figure 5 shows a diagram of the debugger after capturing this scenario with screenshots for the different parts of the application. In the following paragraphs we gradually explain the visual elements of the debugger and the actual debugging process.

Fig. 5. Diagram with screenshots of the debugger at work.

Look and Feel. The 2 clients of the conference application are shown on the right. They are visually extended by the debugger view, by adding a data view at the bottom. It displays the current model of each client. That model contains the information about the room it is displaying, so far it has received the latest version. Additionally, there is a button above the data view, that switches the

⁴ <https://github.com/JeffHoremans/elm-multitier-examples>

interface to the server debugger view as displayed on the left. As indicated by the dotted line, it acts as the main interface of the debugger. Client debugger views can switch back and forth to the server view for seamless debugging.

The server debugging view is composed out of roughly 4 segments, as numbered on the figure: (1) controls for various actions and configuration options, e.g., switching back to the client view, pausing and resuming, showing causal dependencies between events and setting the resume strategy – more about these later. (2) a data view of the server model (similar to the client view). In this example it contains the list of conference rooms and their information. (3) a navigable event list, containing all events from the whole client/server application which make up the global timeline with the most recent event at the top. (4) a graphical representation of this global timeline which shows extra information such as client/server communication in an intuitive way.

Debugging the Scenario: Interface. Looking at the event list (3) and the global timeline of our example (4) we can identify three types of events: initialisation events, message events and RPC events. The first represents the server or a client starting with its initial state. Message events happen when the server or a client program receives a message for the update function and RPC events describe an asynchronous client to server request. Looking at the figure, we can see that the debugger captured these events in our room reservation example. We color-coded events of a single timeline in the event list to make clear which events belong to which timeline in the global timeline.

Before looking at the scenario, we consider type definitions of a few parts of the example application in Listing 1.1. The first line contains the definition of the client model which holds the last-received list of rooms and the reservation name. On the second line, the client’s messages are defined (as a union type). On lines 5 and 6, we can see the definitions of the server model and that of the room type. Finally, on line 7, remote server messages are defined, used in the asynchronous client to server RPCs. Because of the multi-tier setting, the server and client model are defined in the same code base, allowing the room type to be shared by them.

```

1 type alias Model = { rooms: List Room, name: String }
2 type Msg =
3   UpdateRoom Room | Handle (Result Error (List Room)) | OnInput String
4 ...
5 type alias ServerModel = { rooms: List Room }
6 type alias Room = { number: Int, name: String, booked: Maybe String }
7 type RemoteServerMsg = UpdateRoomOnServer Room

```

Listing 1.1. Some type definitions of our example application.

Debugging the Scenario: Identifying a Bug. The example scenario contains a bug where the room is scheduled twice. To find the bug we pause the application and use the event list and the global timeline to inspect the sequence of events. For now, it is sufficient to know that when selecting events on either of them (in 3 or 4), the data view of the server (2) and the data views of the clients output the model state at the time the event occurred.

If we look at the events sequentially, we first see the initialisation event of the server, followed by that of the first client. The latter is also visualised in the global timeline, notice that the client's timeline starts later. Next, the `OnInput` message is received from the DOM indicating that this client has inserted his reservation name. Then, a second client can be seen connecting and doing the same thing. After that, client 1 tries to schedule the room, triggering an `UpdateRoom` message on the client which updates the room with its reservation name. That message causes the `UpdateRoomOnServer` RPC which performs the actual reservation of the room on the server. The `Handle` message received on the client as a response signifies this has succeeded, containing the updated room information. Before client 2 is aware of this, he also attempts the same action and also succeeds. At that point we can see that the updating of the room on the server incorrectly overwrites the room information and does not check if it is already taken.

What this shows us, is that the bug is related to that part in our implementation. The debugger gives us a good estimate that the code that dispatches the `UpdateRoomOnServer` RPC contains a bug. This interactive inspection of the client/server program as a whole quickly allows programmers to identify bugs in sections of code.

4 Multi-tier Elm Debugger

Fig. 6. The Elm multi-tier debugger architecture

4.1 Architecture

The Elmsvuur debugger itself is written as an Elmsvuur application, it follows the Elmsvuur architecture of Section 2.2. To debug other Elmsvuur applications, it wraps itself around them by extending each element of their architecture with its own. This concept is shown in Figure 6 where each element of the debugger architecture is an extension of the instrumented application. It allows the debugger to have complete control over the application.

Both on the server and on the client, the application model is nested in the debugger model. The debugger stores the application’s model as well as additional state specific to the debugger. The top-level messages are debugger messages, so application messages are wrapped as debugger messages and first handled by the debugger update function. This allows the debugger to monitor messages before they reach the application and make it possible to store or discard them before propagating the message to the application’s update function. By wrapping the view function, the debugger is able to add debugger specific sections to the interface.

4.2 Global Timeline

The most prominent feature of the debugger interface is the visualization of the global timeline. It is shown in the form of an event list and in the form of a graphical global timeline.

They both form the same interactive stepper, with each node on the graph (representing an event) in the global timeline corresponding from left to right with the event data in the list above from bottom to top. Clicking events, either in the list or on the timeline, can be used to step the application back to a previous point in the global timeline. There are two functionalities exposed by stepping back in the timeline. First, it can be used to quickly inspect different states of the server model without impacting the entire application. Secondly, if the client/server application is paused its true functionality can be used to step back through the entire application. This results in the server and *each* client updating their data view with their past model at the time of the event. For clients, this also means updating their UI to its past state. Stepping through the timeline thus allows inspecting events, but also shows how the models and views of the different parts of the applications evolved over time. The entire client/server application can be inspected as a single entity.

Event Order. An important thing to consider is the order of the timeline events. We do not rebuild a global timeline based on some notion of synchronized clocks. Instead, the debugger keeps track of a list of events in the order they are received on the server. We make some assumptions to make this practical:

- The global timeline does not give an exact reflection of the real timeline. The global timeline does not allow for events that occur at the same time. From the server’s point of view, an order is enforced.

- We do not visualize the actual time in the timelines, e.g., in reality an event can take a month to occur yet it shows up right behind the last event in the timeline.

We found these assumptions to provide a good balance between reflecting the actual situation and providing a good user experience. While our approach cannot reproduce the exact global timeline, it does give an approximation that has sufficient guarantees for debugging purposes.

Note that our prototype implementation still uses common HTTP requests to send event data over the network to the server from the clients. At the moment this is an implementation bug since the order of multiple HTTP requests cannot be guaranteed to be the same as the order in which they actually occurred. This can be solved by using communication that ensures in-order delivery, such as websockets.

Pausing and Resuming. When the application is running, the models of the different tiers are kept up-to-date and the timeline is being updated with all incoming events. From the moment the application is paused, the interface of the clients is frozen. The debugger then allows quick inspection of previous states throughout the entire client/server application.

When stepping the application back to a previous point in the global timeline, the server and clients are notified to step back to their previous state at that point. For this to be possible, the debugger does not just store each event that occurs, it also keeps snapshots of the model of the server or each corresponding client. With that information, the debugger can find the latest model state for each connected client and the server whenever a point is selected to step back to.

Since we are dealing with a distributed environment, pausing and resuming the application does not happen instantly at every tier. Between the time of pausing at the server, and the client being instructed by the server to pause as well, client events can have occurred. The same applies for resuming the application. As such, in the background the application still runs and reacts to the outside world. When the application is resumed this backgrounded model is made current again and the application can seamlessly continue. This process is shown in Figures 7 and 8.

Stepping Back and Changing the Global Timeline. The debugger not only allows quick inspection of several points in the timeline, it also allows the programmer to resume from a previous point. This can be useful to test multiple related scenarios which take many common events to set up, for instance testing different error cases in a form.

Resuming from a previous point or event in the global timeline and thereby changing it raises some important challenges:

- The debugger should be able to determine which events after the resume point need to be kept and which to be removed. Events triggered by user

Fig. 7. Example timeline when paused

Fig. 8. Example timeline after resuming

interaction or subscriptions need to be discarded as they are exactly the type of events we would like to change by resuming from a previous point. But, for events corresponding to side-effects this does not necessarily apply. In Elm, side-effectful code is always executed asynchronously and their results enter the program as a message. If such messages are the result of a side-effect that started before the resume point, it needs to be kept to keep all the events in the timeline consistent. The same applies for a chain of these connected events that originated before the resume point. An example of this problem is shown in Figure 9.

- By changing the global timeline, at any point in time, events can still be received asynchronously from finishing side-effects (even from past timelines!). Some of these results may be started because of an event that has already been removed from the timeline. These results need to be detected and discarded to keep consistency.

The debugger deals with these challenges using two concepts, parentage and run cycles:

Causal Dependencies: Parentage. To handle the first challenge, the debugger first needs to be able to distinguish between events that were caused by user interaction, by subscriptions or indirectly by side-effects from processing other events. It does so by wrapping message events triggered in the view, by subscriptions or by commands appropriately. This already allows discarding the first two types of events properly. Whenever the timeline resumes from a certain point, all

Fig. 9. Discarding events when changing the timeline

events that originated from HTML (the view) and subscriptions are discarded immediately. But for the last type we need more information to determine if it should be discarded or not. For those events we need to be able to check if they are part of an event chain that originated before the resume point or after. This is where the concept ‘parentage’ comes into play.

For this, the debugger keeps track of direct relations between events. In Elm side-effectful code returns its results asynchronously in the form of a message. By wrapping that message definition with a reference to the event that caused it, it becomes possible to figure out where these messages originated from. Using this reference, we can later identify parent relationships between messages. With this feature, we can handle event chains properly and check if new messages need to be discarded or not. Both cases can be seen in Figure 9 that shows an example discarding process with parent relations visualised.

Since the debugger keeps track of these causal dependencies anyway, we decided to visualize them using arcs in the timeline. In Figure 10, we include a small screenshot of a global timeline with this feature enabled. From our experience, this functionality is useful for users to get a better understanding of how events are related.

Fig. 10. Example timeline with parentage

Discarding Events with Run Cycles. The debugger addresses the remaining challenges by using run cycles. A run cycle is a number which corresponds to a specific instance of the global timeline. The debugger wraps all events with this value.

Whenever a new global timeline is started by resuming from a previous point in time, the run cycle is incremented and the older versions are considered to be invalid. All events from this point on are wrapped using the new version of the number.

Figure 10 shows how we can use run cycles to discard the proper events. The program is run (cycle 0) and is paused before a final asynchronous side-effect has finished computing (pending (0)). We resume the application from a previous point (the second Msg on the client) which creates a new run cycle (1). We now have two pending asynchronous side-effects belonging to two different run cycles. The current run cycle is 1, as such, whenever pending (0) finishes computing its messages are discarded.

Fig. 11. Using run cycles when rerunning events

Invalidating Clients. When changing the global timeline it is possible to rewind all the way back to a point where clients did not yet exist. In this case these clients need to be invalidated since they should not have been created yet. The timeline of such clients is completely removed from the debugger interface and a message indicating their invalidation is shown on the clients. An example can be seen in Figure 12. The client can still be used as an interface to the server debugger interface, but a refresh is required to start a new client session.

5 Future Work

More Efficient Snapshots. The debugger stores data for each incoming event, including a snapshot with the current state of the application. When debugging for long periods of time, this amount of data can grow large and eventually make the debugger too slow. Using the alternative method of recomputing the program state instead of storing it does requires less space, but will grow in computation time instead.

Fig. 12. Screenshot of an invalidated client

The better solution would be to combine them and snapshot the application only occasionally. Inspecting or continuing from a previous event state then only requires recomputing from the last snapshot point.

Importing and Exporting Sessions. The people that encounter bugs are often not the same people who are responsible for resolving those bugs. In most cases the latter are forced to rely on a description of the scenario that led to the bug. Recreating the exact bug is not always straightforward, especially bugs that only occur in rare circumstances.

Being able to export debugging sessions and import them afterwards, would solve this problem. This enables inspecting the exact actions performed leading up to the bug as if it was the original debugging session.

Importing sessions has some requirements though. When importing a session, the program may already be altered, possibly making it impossible to import the session. It would be necessary to detect if the session that is to be imported is still consistent with the current program.

Hot Swapping. The debugger currently does not allow for hot swapping modified code. As a result, checking if a bug is resolved after making changes requires restarting the debugger and the debugging scenario that caused the bug.

Having the feature of importing and exporting sessions, discussed in the previous paragraph, could offer a partial solution. The debugger could then export the debugging session with the bug, restart using the modified program and import the session again for retesting the bug. But this only works if the consistency requirement of importing a session is met.

Client Resurrection. When stepping the application back to a point where currently disconnected clients were still connected, the debugger has no means to restart them. A potential resurrection strategy could be to let the client that is operating as the debugger interface, open a new window and start a special client who will serve as the new host for the old client. But opening new browser windows with JavaScript can possibly be blocked and is currently not natively

supported in Elm. This would also require changing the initialisation phase of the debugger, as currently every new client session starts a new timeline.

A simpler option would be to extend the debugger interface and allow switching to resurrected clients in the same client window, which solves the remarks of the previous approach. The server already keeps track of client connections, so when it detects trying to contact a disconnected client, it can reroute the information to the clients who are operating as the debugger interface.

Visualising WebSocket Communication. Elmsvuur supports two kinds of communication between client-server: RPC communication and WebSocket communication. The first of which is already visualised in the global timeline of its debugger. RPC communication is controlled on both ends by Elmsvuur and is a part of its multi-tier architecture. This allows them to be extended by the debugger to include the parent information that is needed to visualize the communication.

To do the same for WebSockets communication, it also would need to become part of the multi-tier architecture. We use the existing Elm WebSocket library on the client, and a custom written server library on the server, both using regular commands and subscriptions to expose their functionality. A possible solution would be to build a construct that builds on those libraries, similar to the RPC construct building on top of the Elm HTTP library.

6 Conclusion

In this paper we propose a multi-tier debugger, a tool that can be used to debug an entire client/server application as a single entity. It focuses on web applications that are written using multi-tier languages, that is, web applications that originate from a single codebase but are deployed to both client and server. The debugger experience is focused on providing an interactive global timeline for the application as a whole. We propose complex features with an intuitive interface that allows programmers to quickly observe different states of a client/server web application, as well as resume from any point in time.

To make our ideas more concrete we created a multi-tier version of the Elm language. The multi-tier language uses an extended version of the Elm architecture. We created the multi-tier debugger not as a low-level tool that inspects Elm programs but as a high-level multi-tier Elm library that wraps multi-tier applications with instrumentation as needed.

We succeeded in creating a prototype that demonstrates complex features such as being able to inspect different states of a distributed application as a whole as well as resume from earlier points in the history. We identified and solved non-trivial issues such as dealing with asynchronous effects.

7 Related Work

We consider mainly work on time-travel debugging, distributed debugging and specifically, multi-tier web application debugging.

Time-travel Debugging. We can roughly divide time-travel debuggers into two classes. One builds its timeline by capturing the application’s state (e.g. ODB [7]), others capture out and incoming calls to the application (e.g. AIDS debugger for FORTRAN [9] or [6] for SML). Liblog [10] is a replay debugging tool for distributed C/C++ applications. It captures the execution of deployed distributed applications and replays them deterministically. For the browser, Dolos [11] captures out and incoming calls to the application for the browser engine, including user inputs and network callbacks.

The examples of these approaches are implemented as low-level tools and intercept system calls or other low level operations (such as calls to libc). In contrast, Elmsvuur is implemented as a library that wraps its target. Elm, a pure language, allows us to make some assumptions which makes this both easier and elegant. In our implementation we capture both the application’s state as well as the out and incoming calls to the application. Other than just its history functionality, our tool focuses on providing a pleasant user experience with easy-to-understand timeline graphics and other visuals.

Distributed Debugging & Causal Dependencies. Causeway [12] takes a message-oriented approach and builds what they call ‘chains of expressed intentions’. Such a chain is a path of a (possibly distributed) computation, akin to a high-level stack trace it displays the causal dependencies between events and the processing thereof. We employ a concept similar to chains in our debugger to graphically show users which events trigger as a direct result of other events. For example, executing an RPC command on the client graphically shows a connection between that client’s view of the timeline and the server’s.

Multi-tier Debugging. In academic work the Hop debugger [13] copes with both server and client-side executions. Their debugger focuses on providing a single-application overview for all stack traces. Errors and their traces report the correct source-code location regardless of whether the error occurred on the client or the server. Albeit with a completely different debugging experience, similarly to Hop, we treat a client/server application as a single entity to debug.

In industry the Google Web Toolkit (GWT) project is the only multi-tier language that considers debugging. GWT is a Java-based multi-tier language that runs its server side on the JVM and compiles Java to JavaScript for client code. It can run in production mode or in dev-mode. Dev-mode does not compile the client code to JavaScript, instead it runs it as a Java application while propagating changes to a browser application. In this mode Java tools can be used to debug both tiers of an application, however, the client/server application is still seen as two entities to debug.

References

1. E. Cooper, S. Lindley, P. Wadler, and J. Yallop, “Links: Web Programming Without Tiers,” *FMO*, vol. 4709, pp. 266–296, 2007.

2. M. Serrano, E. Gallesio, and F. Loitsch, “Hop: a Language for Programming the Web 2. 0,” *DLS*, pp. 975–985, 2006.
3. A. Chlipala, “Ur/Web,” *POPL*, pp. 153–165, 2015.
4. M. Serrano and V. Prunet, “A Glimpse of Hopjs,” *ICFP*, 2016.
5. M. Serrano, “A Multitier Debugger for Web Applications,” *WEBIST*, pp. 5–16, 2014.
6. A. Tolmach and A. W. Appel, “A Debugger for Standard ML,” *J. Funct. Program.*, vol. 5, no. 2, pp. 155–200, 1995.
7. B. Lewis, “Debugging backwards in time,” in *AADEBUG*, 2003.
8. E. Czaplicki and S. Chong, “Asynchronous Functional Reactive Programming for GUIs,” *Pldi*, pp. 411–422, 2013.
9. R. Grishman, “The Debugging System AIDS,” in *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, AFIPS ’70 (Spring), pp. 59–64, ACM, 1970.
10. D. Geels, G. Altekar, S. Shenker, and I. Stoica, “Replay debugging for distributed applications,” *USENIX*, pp. 289–300, 2006.
11. B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, “Interactive record/replay for web application debugging,” in *UIST*, pp. 473–484, ACM, 2013.
12. T. Stanley, T. Close, and M. S. Miller, “Causeway : A message-oriented distributed debugger Causeway : A message-oriented distributed debugger,” *Source*, 2009.
13. M. Serrano, “A Multitier Debugger for Web Applications.,” in *WEBIST*, pp. 5–16, 2014.