Scaling CAD Software and Enabling Collaboration via Operational Transforms

by

Jonathan Terry

B.S. Electrical Engineering and Computer Science, M.I.T. (2015)

Submitted to the Dept. of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2016 [September 2016]

Author ..... Signature redacted
Dept. of Electrical Engineering and Computer Science
August 16th, 2016

Certified by ..... Signature redacted
Dr. Christopher Terman
Principal Investigator
Thesis Supervisor

Accepted by ..... Signature redacted
Dr. Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Scaling CAD Software and Enabling Collaboration via Operational Transforms

by

Jonathan Terry

Submitted to the Dept. of Electrical Engineering and Computer Science
on August 16th, 2016, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

Across all disciplines, the modern work environment requires one to collaborate with others. Whether it be a news office or an engineering firm, real-time collaboration, specifically through software tools, is becoming the norm rather than the exception. Currently, there are many different software tools available that allow for real-time collaboration. The ubiquitous Google service Google Docs allows for several people to enter data into a spreadsheet or text document simultaneously, greatly increasing productivity. More recently, mechanical CAD tools have been developed that allow for multiple users to edit object models simultaneously. In the same vein as these tools, this research set out to create a robust implementation of collaborative CAD for circuit design, with applications in education.

Within this thesis, I set out to implement a scalable and robust collaborative CAD software through the partial revision of JaDE, an existing circuit design tool. JaDE (Javascript Design Environment) is a web-based software tool used in MIT classes to teach both analog and digital circuit design. In upgrading the software, I set out to achieve several goals, ultimately leading to a lightweight, scalable version of JaDE which supports caching of large static elements and includes a revised database as well as modular schematic updates. Upon establishing these features, operational transforms were then added in order to allow for real-time collaboration. The intended application of this technology is for education, allowing teaching assistants and instructors to help students and demonstrate concepts in a collaborative manner.

Thesis Supervisor: Dr. Christopher Terman
Title: Principal Investigator

Acknowledgments

First and foremost, I would like to thank my research advisor, Dr. Chris Terman. Before Chris was my research advisor, he entrusted me to teach his favorite course and supported me financially, making my chance at a Masters of Engineering a reality. In addition to this benevolent act, Chris later took me on as a graduate researcher in order to complete my thesis requirement in a timely fashion. Throughout my time knowing Chris, he has consistently provided guidance and support. During our periodic meetings he produced excellent feedback and set realistic goals. Aside from business, Chris provided nothing but the best banter, and made the thesis a fun and enjoyable process. While the algorithm to find the perfect acknowledgement may be intractable, let it suffice to say: I would be nowhere without Chris.

The next two people whom I would like to thank are my parents, David and Geraldine Terry. From a young age, they fostered my intellectual curiosities giving me creative license and encouragement every step of the way. As a child my father would design and build toys with me, let me play with his old Apple II, and explain a myriad of scientific advancements. All the while, my mother nurtured my artistic interests by teaching me to cook, appreciate art, and explore the humanities. My life has been immeasurably impacted by my parents, and without them I surely would not be the man I am today.

Last, but certainly not least, I would like to thank MIT's Electrical Engineering and Computer Science department for supporting me as a teaching assistant, as well as Quanta Computer Incorporated for providing a research stipend for the summer.

Contents

List of Figures

Chapter 1

Introduction

In the modern world, collaboration is a necessity in order to accomplish many goals, both inside and outside of academia. A key facet of academia that benefits from collaboration is the education of students. Save the most staunch autodidacts, students benefit from working with their peers as well as their instructors to collaboratively solve problems, with each modality conferring a unique purpose. In this master's thesis, I set out to upgrade JaDE (the Javascript Development Environment) in a manner which allows for greater scalability as well as collaboration between students and educators alike. In order to chronicle the development of collaboration in the realm of circuit design, herein I present the following chapters:

Chapter 1 provides an introduction to the original state of JaDE, its deficiencies, and previous work which suggests ways to improve the scalability as well as allow for real-time collaboration.

Chapter 2 delves into the initial infrastructure upgrades that JaDE warranted, their details of implementation, and measures of performance.

Chapter 3 elucidates the underpinnings of modular schematic updates, the algorithms involved, and measures of performance.

Chapter 4 is a discussion of Operational Transforms in a CAD setting, and the applications of collaborative CAD from an educational perspective.

Chapter 5 includes future directions for research both in software development and education.

1.1 JaDE: The Javascript Design Environment

In order to bring forth a collaborative CAD environment, this thesis will rely heavily on an existing piece of CAD software known as JaDE (Javascript Design Environment). Written in javascript, the aptly named application typically resides in browser windows or iframes and is used in teaching introductory circuit design courses at MIT. With a friendly and intuitive user interface, there is not much left to be desired to the eye (Figure 1-1). However, beneath this unassuming GUI lies several problems with the data model, including the bandwidth consumed by the application. Upon every update, the entire schematic state is sent to the server which for larger labs can mean tens of kilobytes transferred when adding a measly NAND gate. It stands to reason that network traffic can be substantially reduced by relaying smaller messages indicating which component was updated and how. This glaring bandwidth issue and its straightforward solution is taken as a engineering problem, and the next two chapters chronicling more efficient architectures, their implementations, and the resulting improvements in performance.

Additionally, JaDE presents an interesting case study in the development of collaborative software for the purposes of education. Drawing on the concept of Google Docs for collaboration, an interesting addition to JaDE would be the inclusion of a collaborative design mode, whereby users can cooperatively build and test circuits. In the same vein as Google Docs, the underlying technology allowing for seamless coalescence of independently modified schematics would be operational transforms. Herein, the application of operational transforms as applied to circuit design software is explored and tested for robustness, with the prospect of developing educational technologies with a collaborative form of JaDE.

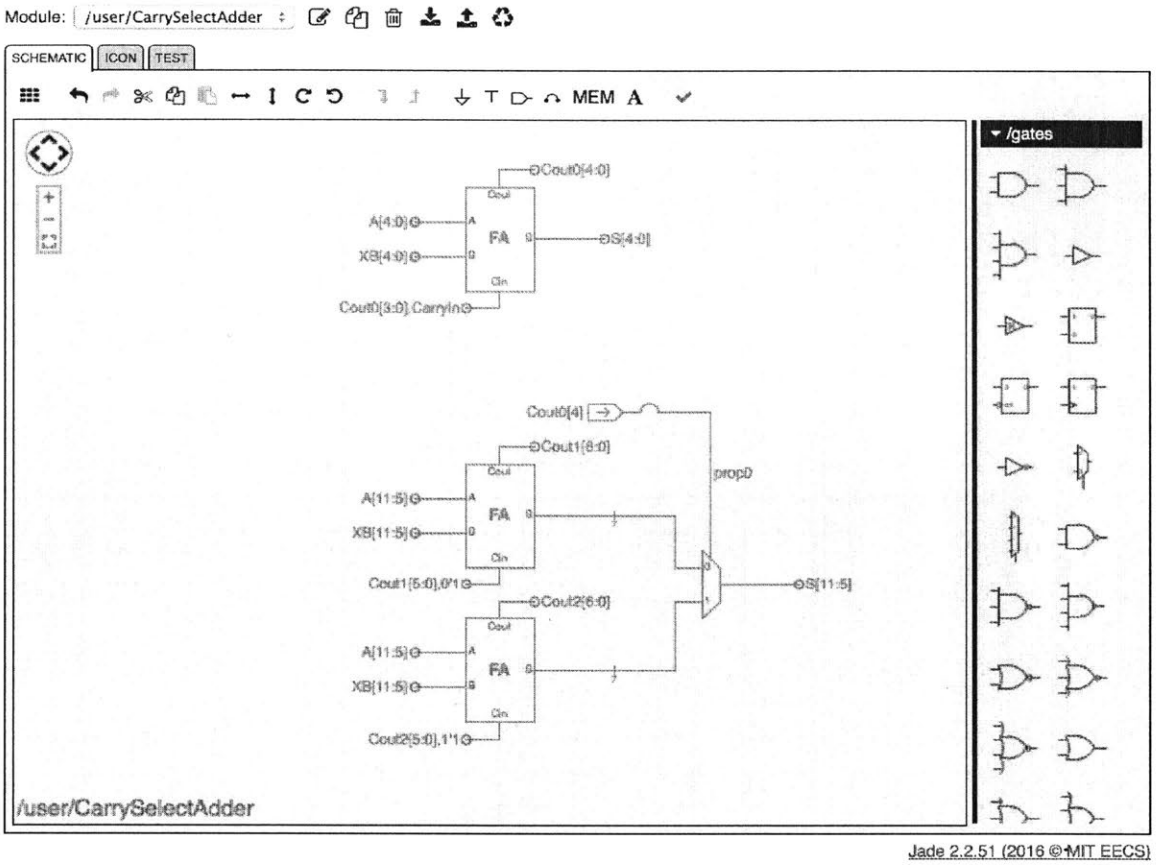


Figure 1-1: Screenshot of JaDE, with its easy to use and intuitive interface.

1.2 Development Stack

With JaDE chosen as the base software for this engineering project, with this decisions comes a pre-defined tech stack that will be worked with. Herein, a brief description of the Mongo, Express, and Node web development stack (MEAN stack, but without Angular) is introduced, highlighting its salient features.

1.2.1 Node.js

Node.js is a Javascript environment which is used to facilitate the development of a lightweight and scalable server [https://nodejs.org/en/]. Node.js supplies a plethora of libraries, which allow for the rapid development of almost any imaginable application. By default, Node comes equipped with libraries to serve webpages over http or https, access the local file system, connect to a database, dynamically generate webpages, and much more. Through the Node Package Manager (npm), the functionality of any web-app can be extended easily. Due its flexibility and scalability, the backend server was built on this technology, and throughout this thesis only compulsory changes will be made to the backend.

1.2.2 Express.js

Express is a concise backend framework offering core functionalities for the Node platform [https://expressjs.com/] . Express in itself is what is responsible for several functions mentioned in the aforementioned section, and facilitates development by being more accessible and easy to use than vanilla Node. In particular, Express is what allows for the easy-to-manufacture entry points in a RESTful API, and allows for the simple, intuitive processing of GET requests and POST requests to the server. This will certainly be used in updating the server to handle modular updates. Although not critical for the thesis, it is also worth noting that Express is capable using templates to dynamically generate and serve HTML to clients. In all, Express is a robust framework which allows for the rapid development of scalable web applications.

1.2.3 MongoDB

MongoDB is a type of database, and a pioneer in the realm of non-relational databases [https://www.mongodb.com/]. Traditional SQL databases are known as relational databases, where the database can be thought of as a giant spreadsheet. In a relational database, all of the information for a given unique identifier is constrained to a row in the database, and column is needed for each attribute. As your web appli-

cations become more dynamic, this form of database can become quite cumbersome to maintain. MongoDB is a non-relational database, where instead of an overarching spreadsheet-like structure there is collection of objects which may even have nested objects or arrays (Figure1-2). In MongoDB, no two documents in the same collection need to even have the same sets of fields, save for a unique identifier. This allows for a more robust and dynamic structure of persisted information, whereby one can access information through the typical object oriented programming dot notation. Additionally, MongoDB allows for objects to have arbitrary field size, thereby freeing the programmer of the restrictions one typically encounters with a relational database.

```
{
        _id: "jdterry",
        L1: [JSON Object],
        L2: [JSON Object],
        L3: [JSON Object],
        L4: [JSON Object],
        L5: [JSON Object],
        L6: [JSON Object],
        L7: [JSON Object],
        DP: [JSON Object]
}
```

Figure 1-2: Depiction of the MongoDB Document structure as an object.

1.3 Modern Collaborative Software

Since the conception of the ARPANET in 1969, a core tenet of computer networks has been the facilitation of communication between people, not just machines [3]. As the

internet gained users, so did the use of email as a communication medium. Although this is not a real-time mode of communication such as talking on a phone, it has had a significant impact on day-to-day communications, to the point where 77% of people find email to be essential to daily activities [1]. In contrast to email, real-time communication over the internet has been developed as well, with services such as instant messenger clients and Google Docs. While the former is usually reserved for personal activities, Google Docs has worked its way into classrooms and businesses alike. Google Docs is a webapp allowing for real-time, collaborative construction of word documents, spreadsheets, and presentations alike. Although collaboration is fantastic in principle, it brings forth the unpleasant realities of distributed systems, especially bandwidth requirements and the concurrent access of data structures.

Behind the scenes, the method by which concurrency issues are mitigated is through the use of operational transforms. All operations require a coordinate or an index, an action to perform at that coordinate, and possibly additional metadata. Operational transforms take modular commands from users, and in effect apply coordinate transforms to try to preserve the intent of the user. This is necessary when dealing with many different editors, as one person's perspective of the nth character could be altered by another person's insertion or deletion. First inspired by Google Wave, operational transforms have been ported to a generalized library known as ShareJS [https://github.com/share/ShareJS], which allow for operational transforms on arbitrary JSON data structures. In order to use this library, it is required that the updates be modular in nature and defined uniquely by some form of coordinate. Piggybacking from the modular updates proposed to fix the bandwidth problem, it appears as if operational transforms can be easily included in a form of Jade utilizing modular updates.

1.4 Previous Work

While there is no standard collborative circuit design software available yet, the idea has not gone without entertainment. Both in academia and more recently in industry have there been examples of forays into Collaborative CAD. With respect to academia, papers and theses dating as far back as the 90's have attempted to implement Collaborative Circuit CAD. Once operational transforms came into fashion, their use in all forms of collaborative software, including CAD, were readily apparent.

In terms of Circuit CAD, a 1997 Masters thesis by Konduri entitled *A Collaborative Environment for Distributed Web-based CAD* details a first foray into Collaborative CAD over the internet [2]. In this VLSI CAD software, the program took the form of a standalone executable to be run in the Java Virtual Machine. This standalone executable, however, was in many ways ahead of its time, resembling what we think of today as a webapp. In the model presented, the majority of the software remained client-side, and the user simply had a GUI by which they issued commands to the host server. Although Jade has a more "client-heavy" design, the overarching distributed systems issues remain constant. In this thesis, Collaborative CAD took a more draconian approach to collaboration, with locks being acquired by users when they want a resource, and then released upon updating the schematic. While this is one way to tackle the problem, thus type of scheme is prone to locking errors as well as divergence of state between machines. All this being said, this work was done before a complete development of operational transforms, but even as such it still provides motivation for this work involving operational transforms.

The year following the submission of the aforementioned thesis, a paper entitled *Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements* by Sun and Ellis was produced, outlining basic algorithms for supporting operational transforms [4]. In this paper, the basic criterion for web-based collaboration is laid out, demanding that for reliable systems one must first insure

convergence of state, intention-preservation, and causality preservation. These terms are explored in greater detail in Chapter 4 of this thesis, but the big picture is that at the end of the day every user must converge to the same exact state, with that state being what the users independently intended design, and that all update messages must arrive at clients in the order they are applied. In order to accomplish this, the authors introduced the Generalized Operational Transform Algorithm, which supports insert, update, and delete operations, with the added benefit of easy undo/redo. Practically speaking, while it is hard to make a provably correct OT system they are still quite useful and certainly far ahead of other technologies for collaborative software.

As the chapter comes to a close, fast-forward a few years to the modern age, and to date operational transforms have had a profound impact on the world as we know it. For instance, Google Docs is a widely used free alternative to Microsoft Office which supports collaboration via operational transforms. Additionally, operational transforms are currently being applied to CAD software in industry: companies such as Onshape now offer real-time collaboration for mechanical CAD software [https://www.onshape.com/]. Finally, in recent years (as mentioned before) a library called ShareJS has been released that allows one to bring operational transforms to any web app they so desire. The increasing popularity of operational transformations and collaborative software vindicate the following research and show a glimpse of the future where collaboration is even more greatly enabled through the internet.

Chapter 2

JaDE Infrastructure and Initial Improvements

This thesis involves a large upgrade to an existing application as well as the backend server. Herein, I discuss issues with the data model, as well as the simple yet powerful ideas which remedied some of the existing scalability problems.

2.1 Motivation for Work

One of the biggest issues with JaDE is the required bandwidth needed to run the application. In its current state, the application sends and receives gigantic blobs of stringified JSON which represent a project in its entirety. This is feasible for smaller projects, considering that we save the project to the server every 25 edits or so. Then consider that for larger circuit layouts (such as a computer processor), these stringified JSON objects may reach hundreds of kilobytes in size, and for slower/mobile connections this can prove quite problematic. Down the road, should the switch be made to alternative hosting services, this could result in higher monthly service charges. In order to tame the wild bandwidth hog, I first restructured the data model and implemented caching aimed at decreasing bandwidth. In this chapter, I describe the data schema, straightforward methods for mitigating bandwidth consumption, and set the stage for the development of modular updates.

2.2 Value Schema for JaDE Data

For a given JaDE project, there is a single value JSON object which contains several different modules comprising the whole circuit. Each of these modules represents a block in a circuit, and contains four separate aspects: the Schematic, the Icon, the Properties, and the Test. Beneath is a graph showing the tree structure of the projects and how one can traverse the tree to address components. Many of these Aspects are seldom modified, and moreover, are rather large. As a consequence, this motivated modifications including client-side caching, a first pass at modular updates, and restructuring of the database.

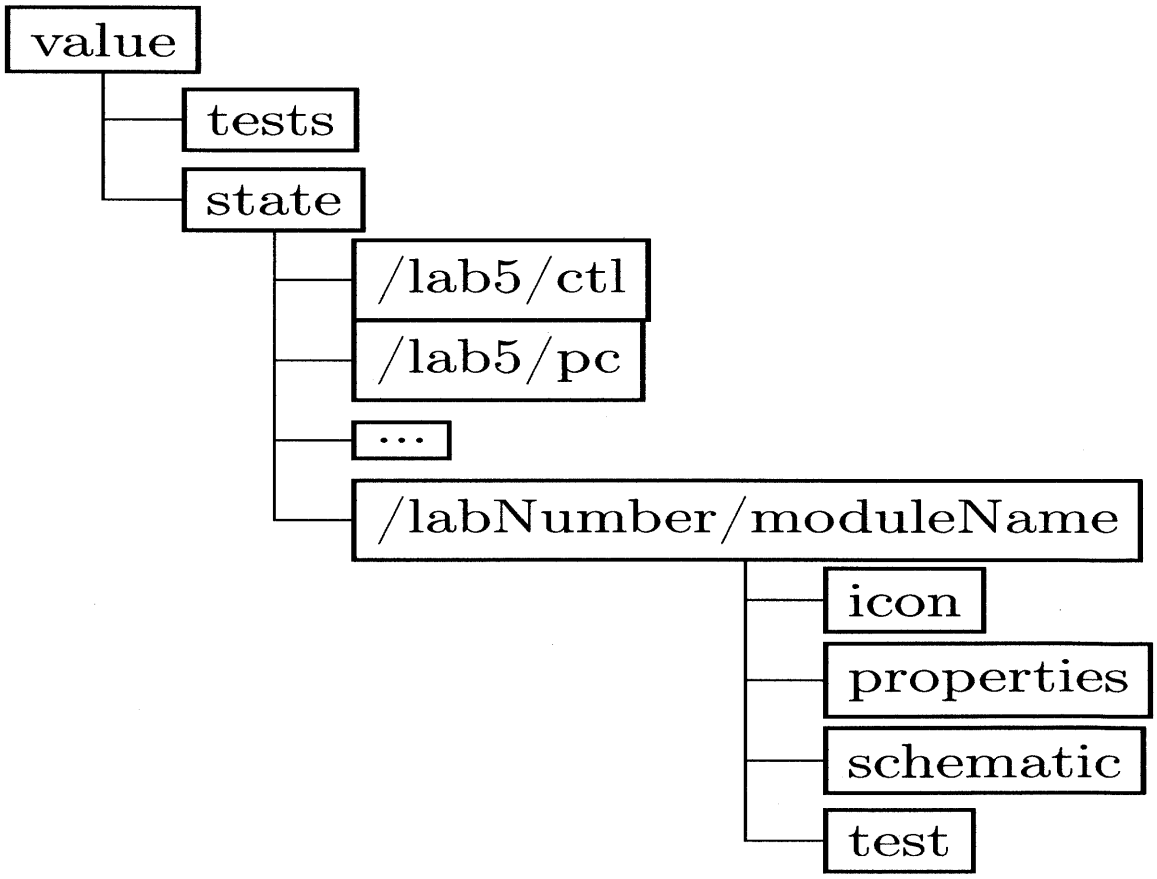


Figure 2-1: Schema of a Value purposed for JaDE data. Note that within a given lab, there are several different modules, each of which has a State containing the four Aspects of Icon, Property, Schematic, and Test. The Tests field which is a direct child of Value stores whether or not the State's Modules have passed their tests.

2.3 Client-side Caching

For a given lab, there is a large amount of initial state information transferred between the server and the client. Every initial module is transferred through the HTML file, including a mess of Aspects which are read-only and static. In the case of larger projects, such as the Beta Processor, the HTML file is around 280 KB with around 270 KB of that being these Aspects. Given the nature of the data being transmitted, I was able to use an MD5 handshake scheme in order to reduce the server load by roughly 99% upon loading the webpage.

In the MD5 handshake system that I implemented, instead of the server transmitting the large chunks of data on first loading the webpage, the server would instead transmit the MD5 hash of the stringified JSON of each Module's Aspects. Upon loading JaDE, the client then checks localStorage for the MD5 hashes, and if present, packages them up as expected and passes them to the JaDE initialize function. In the event that the hashes are not present in localStorage, either because the initial state has since been updated or it is a person's first time visiting the page, then it contacts the server with the missing hashes, and if applicable, clears up any stale objects in localStorage. As a result, there is much less to load on each return to the website, thus reducing latency as well as network traffic.

Although the performance improvement appears to be quite dramatic, it is important to remember that this improvement is only applicable when reloading the lab. For instance, if a person just visits the lab url once, they gain almost no benefit from this upgrade, except for shared modules which may exist across labs. Fortunately this is not likely to be the case, and as a result may be a useful addition, especially in terms of increasing the scalability of the server. It also served as a good first foray into the existing code base, as it required upgrades to both JaDE and the server.

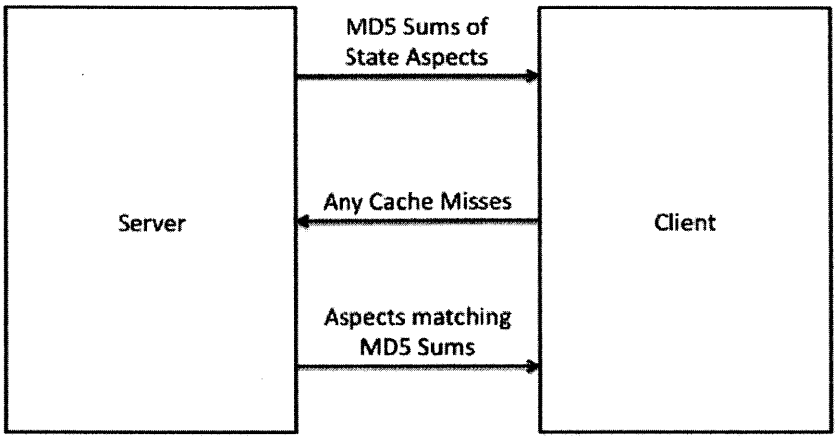


Figure 2-2: Diagram depicting the interaction of the client and the server with the upgraded handshake system.

2.3.1 Server-side Upgrades

On the server-side, there were two main updates that had to be performed: restructuring the build process as well as creating a new route to handle the MD5 handshaking.

Build Process: In the build process, the HTML that is served needs to now contain the MD5 hashes instead of the huge chunks of stringified JSON. In order to accomplish this, the build process has been restructured in a two ways. First, the Module Aspects are now centralized into a single file named `repo.json`, from which the MD5 hashes are derived. This allows for greater consistency across different projects, as the previous method had manual entry for each lab project. With a central repository, the MD5 hash will be guaranteed to be the same for each reference to a certain module, allowing for a less cluttered localStorage!

Building the repository was simple enough, and took the form of a JSON file with `Module.Aspects` as keys (for example, `/lab1/and2:schematic`) and the JSON for the associated Aspects as values. Then, I simply changed the python build process to figure out which Modules were necessary, then replace the information in the answer tag with mappings of `Module.Aspects` to the associated MD5 hash derived from the repository. Since the original html files are left intact, simply swapping the build scripts would allow one to revert to the previous non-caching system should the need ever arise.

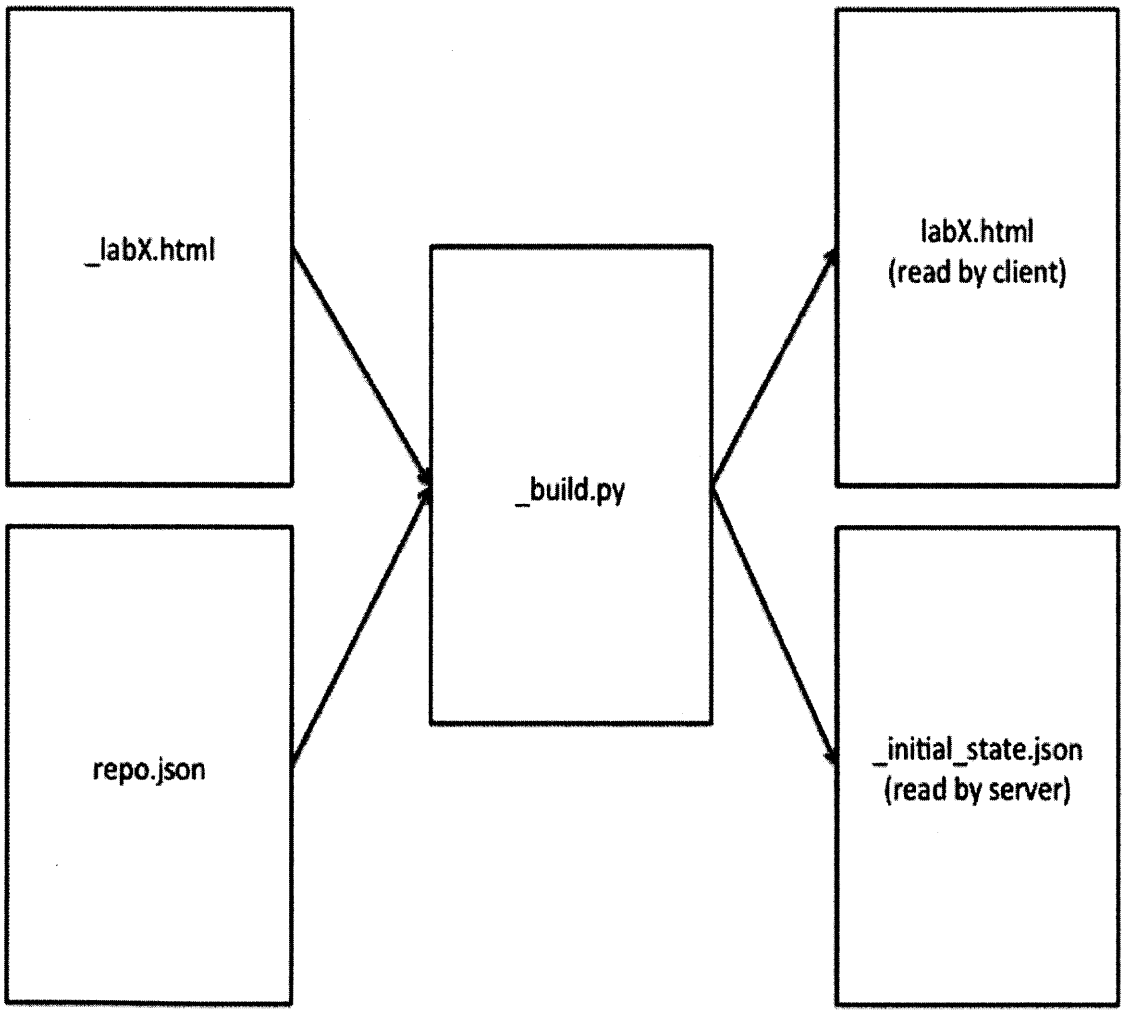


Figure 2-3: Diagram depicting the interaction of the static initial states with the existing lab infrastructure to produce lightweight site content as well as a JSON object for the server to serve aspects from given an MD5 request.

Server Route: Additionally on the server side, a new route was needed to serve the large static content. This route simply is an extension of the user route, and takes the hash as a parameter for a GET request, returning the stringified Aspect as a response. The Aspect to be returned is simply looked up in the JSON object that is created in the build process. To facilitate a clean RESTful API, one will note that Aspects are requested one at a time by the client. Should the need ever arise to revert to the previous system, this addition would simply be irrelevant, and could simply exist happily without ever doing anything.

```
var initial_states = JSON.parse(fs.readFileSync('labs/
    _initial_state.json'));
router.get('/initial_state/:hash', function(req, res, next) {

    var return_object = {};
    return_object[req.params.hash] = initial_states[
        req.params.hash];

    res.json(return_object);
    res.status(200);

});
```

Figure 2-4: Code listing for MD5 server route, highlighting the modular interaction with the new module repository created by the build process.

2.3.2 Client-side Upgrades

In terms of the client-side upgrades, I created three major functions in an auxiliary file named `jade_thesis.js` which contains helper functions pulled into the minimized JaDE file used in production. The three functions that support the bulk of the work were as follows:

1. `jade.server_client_sync(md5_config)`: Synchronizes read-only material in localStorage with the master copies on the server.

2. `jade.request_initial_state(missing_aspects, callback)`: A function which iterates over the list of missing aspects, making the appropriate ajax calls to the route mentioned in the previous section.

3. `jade.construct_initial_state(md5_config)`: Wires together the cached data in the way that JaDE expects.

4. `jade.collect_trash(md5_config)`: Cleans up the localStorage to prevent clutter and stale data from being cached.

For the purposes of illustration, below is a code listing of the `jade.server_client_sync` function which succinctly conveys the algorithm. For further information on the other functions at play check the documentation in the appendices.

```
jade.server_client_sync = function(md5_config) {

    if (!this.detect_storage) {
        throw new Error("Unable to invoke localStorage, please
            use an HTML5 compatible browser.");
    }

    var missing_aspects = [];

    for (module in md5_config) {

        var module_object = md5_config[module];

        for (aspect in module_object) {

            var hash = module_object[aspect];

            if (localStorage.getItem(hash) === null) {
                missing_aspects.push(hash);
            };

        };

    };

    if (missing_aspects.length > 0) {
        this.request_initial_state(missing_aspects,
            this.update_local_storage);
    };

    this.collect_trash(md5_config);
    this.construct_initial_state(md5_config);
};
```

Figure 2-5: Code listing for client-server synchronization algorithm, making use of caching.

2.4 Restructuring the Database

One final upgrade in this phase of the thesis, was conversion of the existing Answers database into a form more conducive to modular updates and eventually operational transforms. In the script, it is use Answers as a template, and creating AnswersX as the resultant restructured database. In restructuring the database, I simply converted all of the stringified JSON into parsed JSON structures. As a side effect, the load procedures for the JaDE data model had to be retrofitted to support this. Below is how the script is used for the conversion, which is simply a redirection to mongo as mongod is running. The complete code listing is in the script appendix, and could serve as good boiler plate for similar operations.

Usage:

```
user@machine$ mongo < reformat.js
```

2.5 Results and Discussion

For this first part of the project, the server and JaDE were upgraded to employ client-side caching. With these upgrades, I then considered the transmitted information on page loads for the circuit design intensive labs. On the next page is a comparison of performance with the two schemes. Noting the log scale on the graph, one can see that performance of loading static content was improved by a factor of 100 for the largest labs.

Additionally, a point of discussion is the load time of the JaDE application as a result of the new client-side caching policy. In terms of application startup time, under both systems, the startup takes around a second after opening the window to be able to use the program. Therefore, the benefits of significantly reduced bandwidth on successive page loads came at no detriment to the speed of the program.

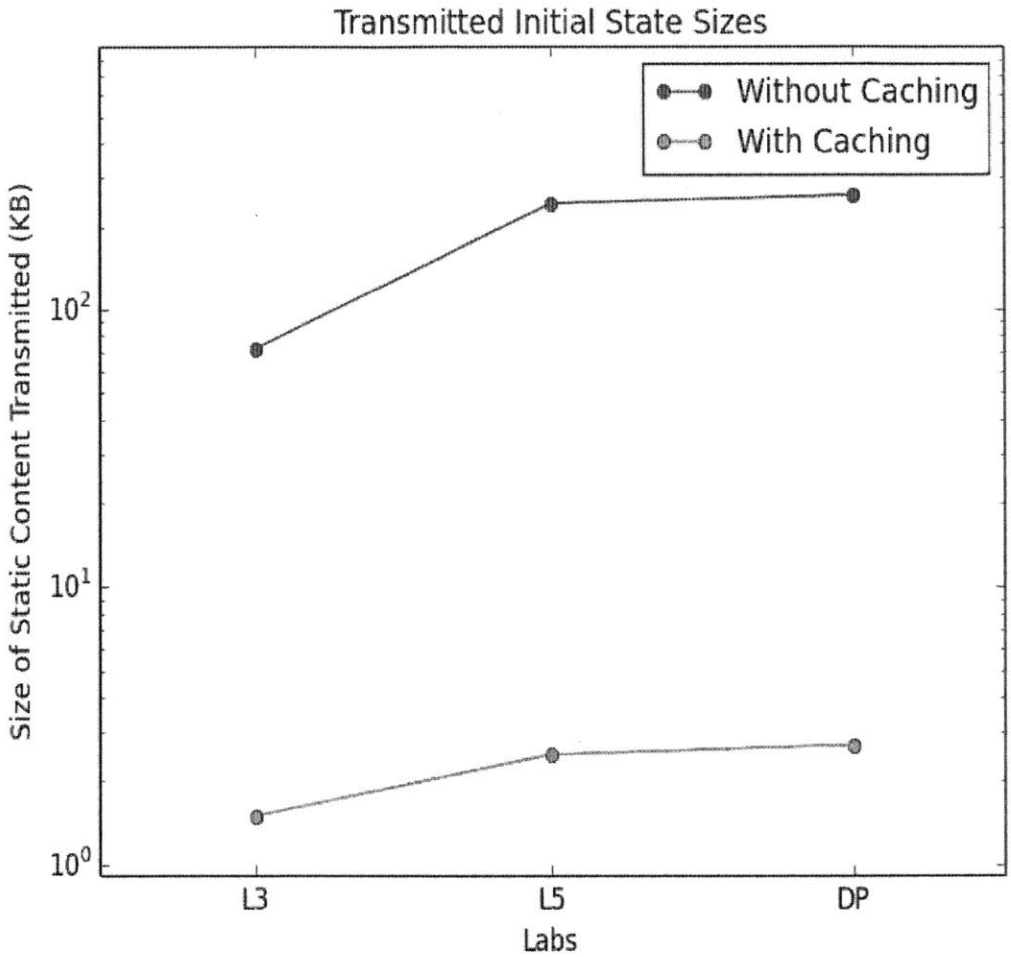


Figure 2-6: Figure showing the reduction in JaDE content transmitted on each page load. From the graph, one can see that the content transmitted is reduced by two orders of magnitude on successive page loads.

Chapter 3

Reducing Bandwidth through Modular Updates

3.1 Overview and Motivation

Building upon the initial foray into reducing bandwidth via a caching scheme, modular updates seek to further reduce the bandwidth used by the application. With modular updates, instead of sending over an entire stringified schematic every few times you place a component on the schematic, the application sends significantly smaller update commands to the server in order to update the data model on the server. Through clever integration of AJAX calls and a message handler class, much of the existing code was leveraged to produce the intended result while allowing for reduction in message transmission size by factor of roughly 500 to 1000 depending on the components updated. Furthermore, in this system, the interfaces and data produced are immediately compatible with operational transforms as supported by the ShareJS library.

3.2 Architectural Changes

In the existing application, there were communications between spawned windows, the host page, and the server. Windows and the host page would send messages to each other via the `window.postMessage` cross-domain communication function. To be more precise, these messages were stringified blobs of state identical to the data model you find in the Mongo database. Upon submitting a schematic update or running a series of tests in a Jade window, the window stringifies its state, and sends it over to the host page which then relays the information to the server if it detects a change. Communication goes the other way as well, with the server supplying information to the host page which then is relayed to the Jade window (see Figure 3-1). Furthermore, outside of Jade, the host page is responsible for relaying any submitted answers to the server. While this architecture is perfectly sound, there is a large amount of wasted bandwidth which needs to be tackled due to transmission of the entire stringified state.

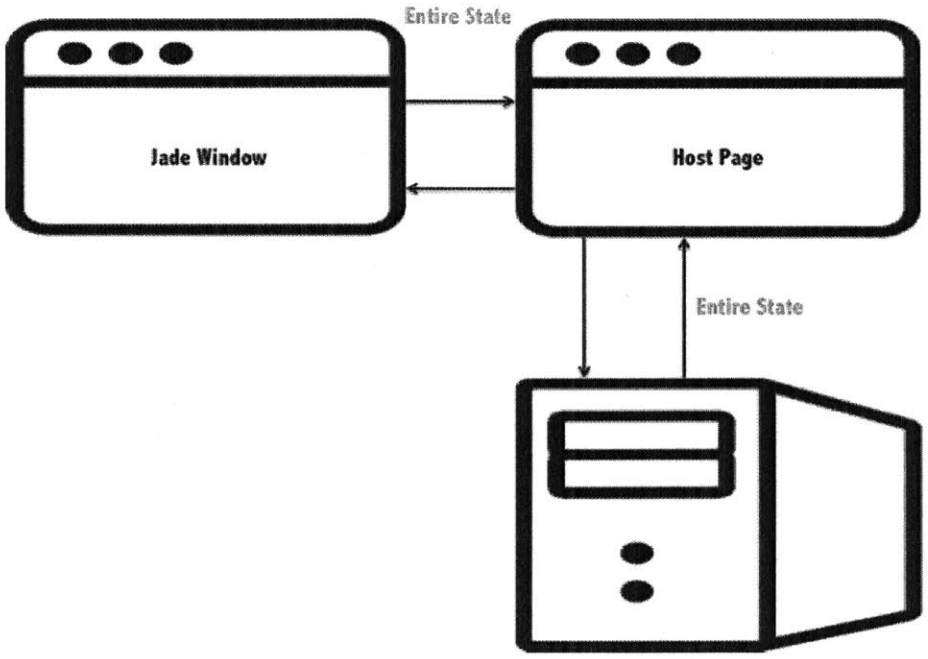


Figure 3-1: Block diagram illustrating how the window, the host, and the server communicated before the modular updates. All communication takes the form of sending entire blobs of stringified JSON state.

In supporting modular updates, the architecture was slightly changed, with the design considerations coming from the fact that the host page only truly needs to relay Jade test information to the server, and not everything else. Upon application startup up, the server still relays state data to the host, which then passes it to Jade for initialization via postMessage. As the user then edits a Jade schematic, instead of relaying information to the host and then to the server, the Jade window communicates directly with the server via AJAX, sending small JSON objects on the order of a few bytes, representing component-by-component changes to the schematic. Additionally, the Jade application may request status updates directly from the server, so that the applications knows it has the most current representation of data. Noting the host page's answer processing ability, the labs script was moidified as to only send test results to the server upon a Jade window posting a message. Figure 3-2 lays out how all of the systems communicate with each other. In all, while there has been some decoupling of various services, the goal was to make the changes as transparent as possible and facilitate real-time communication between the server and the application window.

In addition to the modular updates which were the most salient upgrade added, this architecture also allowed for the implementation of a user interface which will hopefully prevent fewer data management issues in the future. With the synchonization data being passed back from the server to the application, the application now alerts the user if there has been a communication error, if there is an inconsistent state detected, or even if there is a second instance of Jade opened somewhere which could lead to an inconsistent state. In addition to these warnings, the application now has an indicator showing when there are messages which still need to be transmitted.

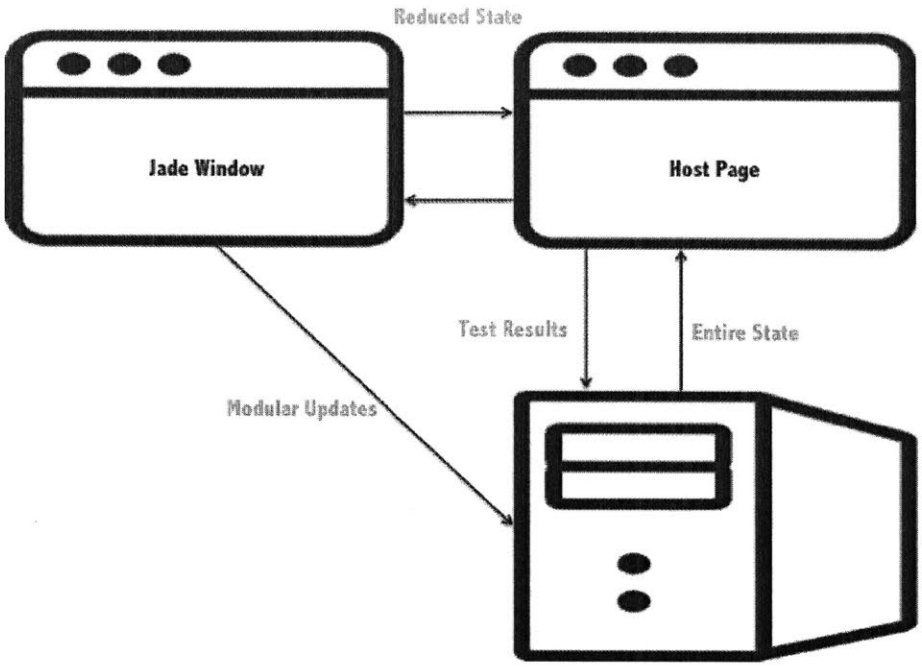


Figure 3-2: Block diagram illustrating how the window, the host, and the server communicated with modular updates. Ways of communication are inherently unequal, which allows for much less bandwidth usage.

3.3 Jade Application Window

3.3.1 Registering Component Level Updates

The first step to making modular updates a reality is to devise a scheme by which components may be referenced without ambiguity, and from there be able to tell when components are either being inserted into the schematic, updated, or deleted. In the existing data model, individual components in a schematic are represented as a tuple `(type, coordinates, properties)`. These component tuples are then aggregated into an array called Components. To tackle the first problem, I exploited the fact that the components are held in an array, and naturally for this type of a data structure and unambiguous reference is that of the index in the array. Therefore, the unique identifier can be given upon adding the element to the components list since upon introduction to an array, the index of any newly added element is simply the length of the array. Furthermore, updating the data for an existing element has no bearing on

the index. The only edge case to consider are deletions, which may require refactoring indices in the array if you delete at any index except the very last. In summary, the types of modular updates as well as their algorithmic complexities are as follows:

1. **Insertion:** Insertion to a schematic merely requires appending to the existing Components array, with the calculation of the index and insertaion running in amortized $O(1)$ time.

2. **Update:** Updating a component in the schematic merely requires accessing an element of the Components array and then changing the data there. Therefore, an update runs in amortized $O(1)$ time.

3. **Deletion:** Deletion requires removing an element from the Components array via the splice function, and then reassigning the indices of all components there after. Therefore, a deletion runs $O(n)$ time, although practically one expects better performance due to temporal locality of associated updates and spacial locality within the data structure.

With the scheme in place by which to reference components, we are then left with the challenge of registering said updates. Fortunately, the model within Jade is structured already such that components are either inserted via `Component.add`, deleted via `Component.remove`, or updated via `Component.update_properties`. With these entry points corresponding nicely with the notion of the aforementioned modular updates, each of these functions simply required a function shoving the modular update data to a queue for transmission to the server. The way by which these component level updates were registered was through the creation of the following method, in which the updates are given a particular type, a path in the data model where which they should be inserted, and actual component data itself. As one can see in Figure 3-3, this modular update now references a new class, which is the message handler, and the workhorse responsible for communicating with the server.

```
function send_component_update(operation, path, component_update)
    {

    var modular_update = {
        update_type: operation,
        update_path: path,
        update_data: component_update
    };

    jade.message_handler.enqueue_request(modular_update);

};
```

Figure 3-3: Code listing for the model's way of registering updates with the server.

3.3.2 Handling Component Level Updates

In order to aggregate all of the component updates and ensure proper communication with the server, a message handler class was created, which exposes the following API to facilitate ease of use:

1. `establish_connection()`: Sends a synchornization message to the server, and based on server response there will generate a message box should state issues arise or if you have a Jade open elsewhere.

2. `get_tip()`: Returns a string indicating message queue status.

3. `enqueue_request(update)`: Takes a server request and prepares to send it to the server.

4. `push_changes()`: Sends the whole queue to the server, generating a message box should there be an error.

Upon wishing to edit a document, one first establishes a connection with the server to insure that the state is kosher. From there the tip is updated as the data model and connectoin status change, which is relayed back to the user through GUI updates. Whenever one wants to update the data model, the request needs to be enqueued. Furthermore, the ability to push changes is exposed so that any object may request

pending updates to be pushed to the server. By default, a non-empty message queue is pushed every 10 seconds or when the mouse leaves the Jade window.

Through this message handler, we can see updates being sent in small groups from the nodemon logs, with significantly smaller update sizes being at least two orders of magnitude smaller than before. Figure 3-4 shows the nature of these updates in action, and a complete code listing may be found in Appendix C.

```
[18/Jul/2016:17:29:21 +0000] jdterry POST /user/lab/L3/2 200 8.308 ms - 42
INSERT applied to /lab3/shift:schematic
UPDATE applied to /lab3/shift:schematic
INSERT applied to /lab3/shift:schematic
[18/Jul/2016:17:29:30 +0000] jdterry POST /user/lab/L3/2 200 2.239 ms - 42
UPDATE applied to /lab3/shift:schematic
INSERT applied to /lab3/shift:schematic
[18/Jul/2016:17:29:40 +0000] jdterry POST /user/lab/L3/2 200 1.324 ms - 42
DELETE applied to /lab3/shift:schematic
DELETE applied to /lab3/shift:schematic
DELETE applied to /lab3/shift:schematic
```

Figure 3-4: Highlights the modularity of updates and how it is processed by the server. The response size of 42 bytes is the returned synchronization data upon supplying a set of updates.

3.3.3 Updates to Entire Modules

While the vast majority of the updates which are necessary involve component updates to a given aspect, it sometimes is necessary to create, delete, update, or copy entire modules. Building upon the primitive updates implemented before, these module-level updates required few lines of code to implement since they rely upon existing modular updates after creation of the module data structure, which is just a collection of aspects. The additional types of updates that were added were as follows:

1. **Module Edit:** Responsible for creating a new module if necessary, otherwise, just switches to an exisitng module of the supplied name.

2. **Module Delete:** Responsible for deleteing a given module in the database, as referenced by a key.

3. **Module Copy:** Copies everything in a given module to a new module, by first creating a blank JSON object for the module.

4. **Reset:** Erases work for all required modules. Note that any user defined modules remain intact.

3.3.4 State Coherence Mechanisms

The final piece that was added to the Jade window was the ability to detect issues related to server state being coherent with with the state currently in the window or the state to be in a certain instance of Jade. As far as it comes to resolving conflicts, I opted that in the event of outdated state, the server state is fetched and taken as canon. The major upgrades which were implemented in order to maintain state coherence are as follows:

1. **State Status:** Jade is now capable of alerting the user as to the state status, whether the server and the client are synced, there are changes client-side that have not been written to the server, and if there is a problem contacting the server at the moment (Figure 3-5 and Figure 3-6).

2. **Multiple Instance Warnings:** Upon connecting with the server, Jade registers with the server that it is open and attempting to edit a certain lab answer. If there is more than one active user, Jade gives a warning message that there are multiple open windows editing a single lab answer (Figure 3-7).

3. **Inconsistent State:** Should one reconnect to the server after there have been changes (whether it be to a network outage or even a power suspension), so long as the server has a more recent state, it will alert the user, and force the client state to become that of the server (Figure 3-8).

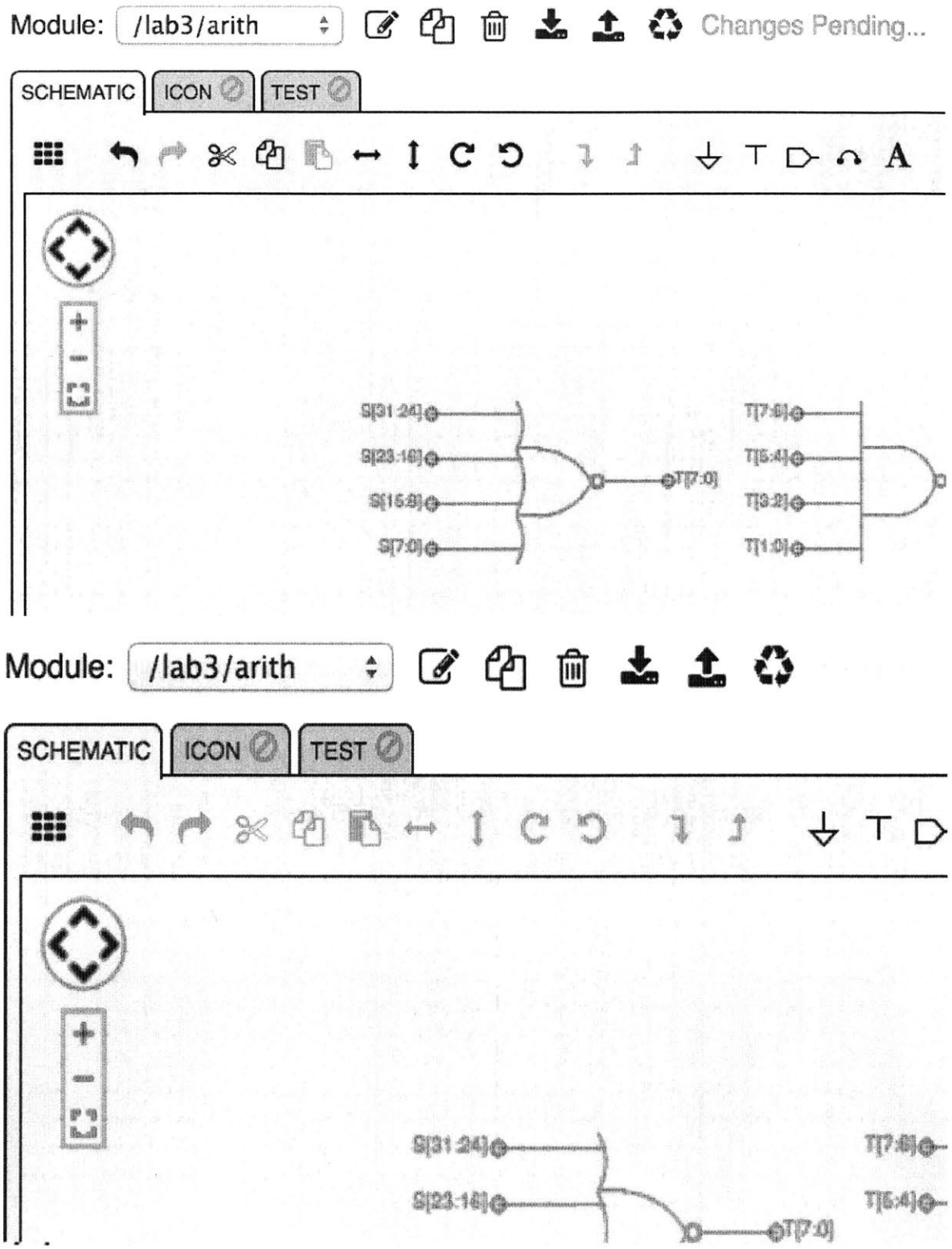


Figure 3-5: In the top frame, there were pending changes to be incorporated by the server. In the bottom frame, the server has acknowledged the modular update and as a result the satus has been changes to synced.

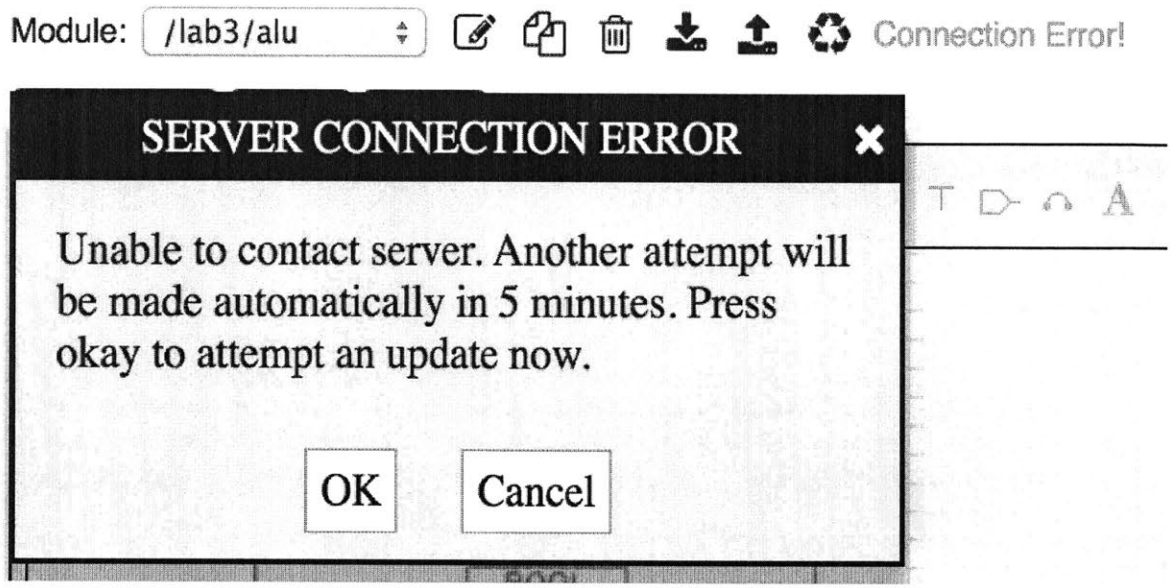


Figure 3-6: In the event that the server cannot be reached, the status indicator changes as shown above and a dialog box is opened informing the user of the problem.

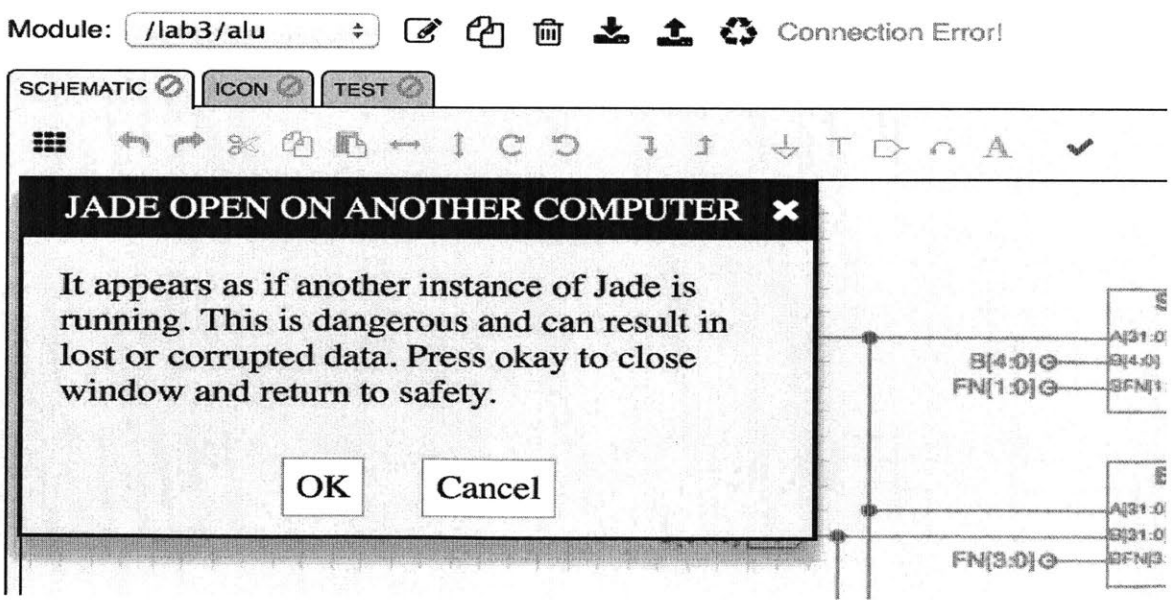


Figure 3-7: Should there be more than one Jade session attempting to edit the same schematic, the user is show the above error.

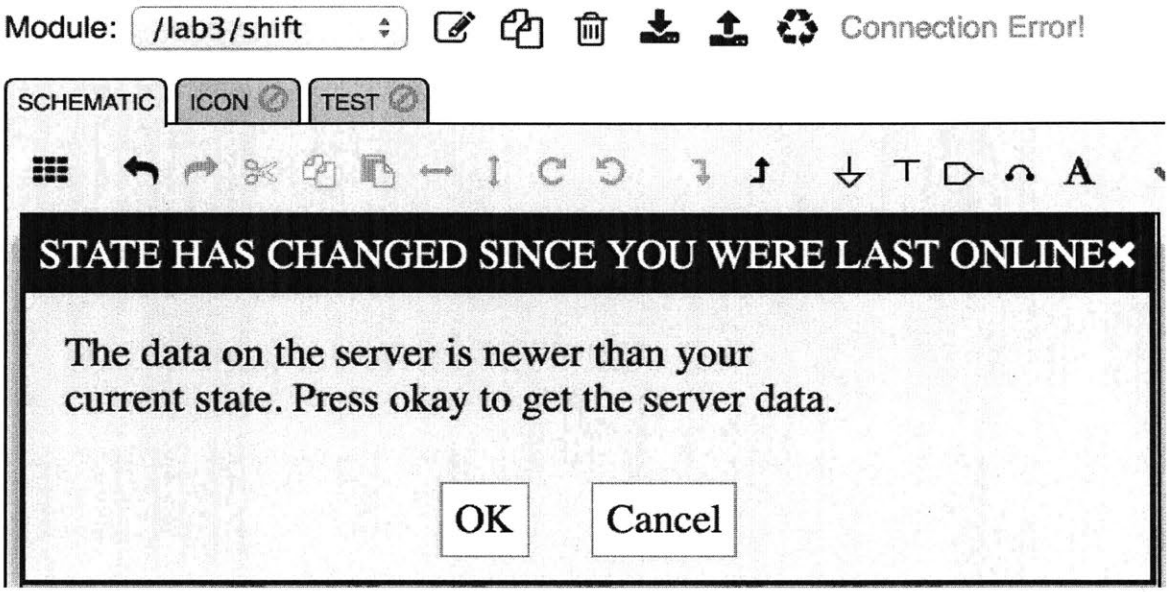


Figure 3-8: Should the current user's copy be out of date, the server alerts them to the problem. Will need to be surpressed/updated when operational transforms are added.

3.4 Host Window

In addition to the server and the Jade application window, there is the host window from which the majority of the lab material is provided. In addition to the lab material, the host window plays a role in maintaining state and communicating with the server. In the previous model, the Jade window sent all of its state to the host, which then relayed it to the server. In the new model, Jade communicates directly with the server, however, the host page answer posting system is still needed to handle answers from non-Jade lab problems. Therefore, the host application has been revamped to only transmit non-Jade answers to the server and results of locally run Jade tests. This makes the function of the host page slightly decoupled from Jade, and has made it such that the job of the host page is to primarily relay complete answers. Additionally, the host page is responsible for maintaining (but never transmitting) the entire Jade state, so that if one decides to close the Jade window, it can be opened again without having recontact the server for data.

In summary, the host window has mainly become a medium by which Jade is initialized from the server state, through which completed answers flow.

3.5 The Server

With the significant upgrades to the Jade window also came rather significant yet transparent upgrades to the server. In total, there were three routes involved in the upgrades:

1. `GET/lab/sync/:lab/:problem/:command` This route is used to get synchronization information from the server, which Jade can then use to figure out if there are any state coherence problems on the horizon.

2. `POST/lab/:lab` This route is now only responsible for keeping track of completed tests and answers. The only real change is that some logic was added upon submission to figure out if it is a jade submission, and change how the database is accessed.

3. `POST/lab/:lab/:problem` This route is the workhorse of modular updates, which accepts a list of updates, and applies them in order to the database representation of the jade schematic.

3.5.1 Synchronization

Synchronization and state coherence is mediated through GET requests to urls of the form `/lab/sync/:lab/:problem/:command`. For a given lab, one can make the following synchronization requests for a given lab answer:

1. `OPEN` Open alerts the server that a new window is trying to edit the given lab answer.

2. `CLOSE` Close alerts the server that a window editing the lab server has disconnected.

3. `RESET` Reset allows one to reset the count of open windows in the event a disconnect is inappropriately logged.

```
var sync_data = {
  generation: generation_number,
  open_jade: number_of_open_jades
}
```

Figure 3-9: Data returned upon a synchonization request.

3.5.2 Answer Posting

With the aforementioned update to the host page script, the server was also updated appropirately to make sure that answer data is stored correctly. Before modular updates, the route was used to accepting a huge object describing all answers for the lab. Now, answers are sent over one by one, and as a consequence, logic was added to point to the correct path, and update the answer. In Figure 3-10, we see the simple logic that was added to update functionality of the route.

```
if (a.value.tests){
    answers[a.id].value.tests = a.value.tests;
    lookup = {};
    lookup[req.local.lab + '.' + a.id + '.value.tests'] =
        a.value.tests;
} else {
    answers[a.id] = a;
    lookup = {};
    lookup[req.local.lab + '.' + a.id] = a;
}
```

Figure 3-10: Logic controlling the proper update of answers based on whether they originate from the host page or a jade window.

3.5.3 Modular Schematic Updates

The big kahuna itself, modular updates were made possible by creating a route which which posted to, would tease through a list of modular updates and apply them to the data model. In a modular update (as seen in Figure 3-3), the route is suppliled with a path indicating the module and the aspect to be updated, the type of operation to be performed, and any necessary component data. From there, the route iterates through the list and applies the updates. Additionally, the generation number is updated after all updates are applied. A complete code listing can be found in Appendix D.

3.6 Results

Last, but certainly not least, it is time to explore how well these modular updates tamed the wild bandwidth hog. My methodology for testing was to run through every straightforward lab containing Jade code, and monitor the network traffic and total information transmitted. I opted against testing the Design Project, since it will vary greatly from person to person. At the end of the day, after running through the labs, I was able to produce the following data in Figure 3-11, which shows that bandwidth was reduced by a factor of nearly several hundred for every lab.

While it is important to note that I have had significant experience with these labs, and that the average user may not use the same absolute bandwidth as I, the salient feauture of modular updates is that it will not matter. Regardless of user proficiency, they will still see a significant reduction in their bandwidth consumed. Since we are now dealing on the order of tens of kilobytes, the usage of jade in the worst case is no worse than loading a facebook page. In all, the modular updates proved to be quite successful in reducing bandwidth. Furthermore, given the indexing system put in place, integration of operational transforms should be rather transparent.

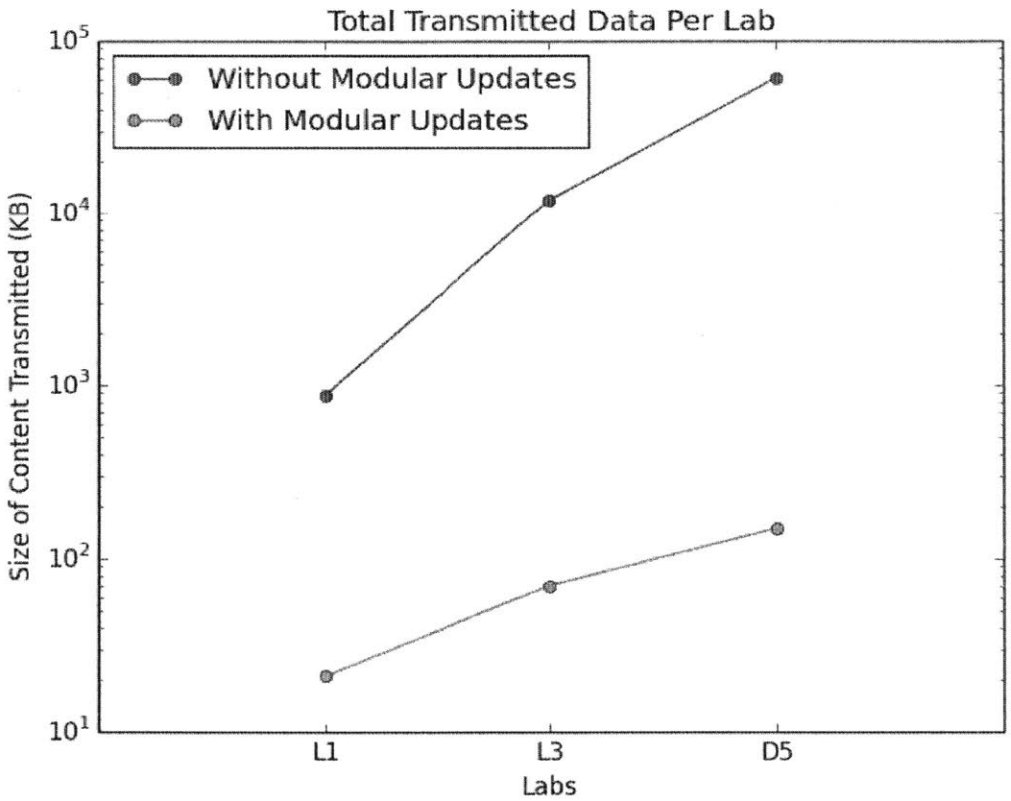


Figure 3-11: Plot comparing the data transmitted during completion of each lab. With the implementation of modular updates, we see that there is a significant reduction in bandwidth.

Chapter 4

Sockets and Operational Transforms: Theory and Practice

Thus far in the thesis, most work has been systems engineering and testing of the resultant engineered system. At this point, the thesis breaks free of traditional engineering work and delves into the realm of computer science research. In the beginning, the theoretical underpinnings of operational transforms are spelled out in simple conceptual terms, and then more rigorously in a formal mathematical way. After preliminary tests, it was determined that in lieu of ShareJS, a custom implementation of the generalized operational transformation algorithm would be implemented. The custom implementation of the generalized operational transform algorithm requires the inclusion of an update command, and instead of working with a string, it is extended to work with any linearly addressable data structure. Upon building the mathematical formalism of the algorithm, its implementation is tested and developed into useful tools.

4.1 Introduction to Operational Transforms

Before one can introduce Operational Transforms, it is worthwhile to understand why they were developed in the first place. Around two decades ago, when collaborative distributed computing was becoming more of a reality, three fundamental qualities were defined for making usable and robust collaborative software in which several collaborators edit a single document:

1. **Convergence:** The idea that after any set of actions from any number of collaborators, each collaborator's local document state is identical.

2. **Intention Preservation:** After applying any set of actions from any number of collaborators, the resultant document state should be what the authors intended.[1]

3. **Causality Preservation:** Recipients of updates from third party collaborators must apply them in the order they are applied by their authors.[2]

With these considerations in mind, let us consider the situation where we have two editors attempting to edit a string, using a naive approach. In this example, let us consider the situation where Red is attempting to insert characters to the front of the string while Blue is attempting to delete characters from the front of the string. Since we live in the real world, there is a finite delay between the two edits, and in Figure 4-1, we see how they would be applied to an example string. As one can see, Blue ends up with a correct version of the string while Red is left with an incorrect state. This stems from the fact that when Blue specified its delete update, it was using the original string's indices to make the specification. As Red updated his string, the intention of the original delete was lost. While this was the case here, operational transforms

[1]At this point, one may ask how intention preservation differs from convergence. While the difference is subtle, it is important. Consider a scenario where every person could converge to a wrong state; this would satisfy the first condition, but violate intention preservation. Likewise, every person could have a copy of the data structure which preserves intention, while being different up to a permutation thereby violating the convergence criterion.

[2]Causality preservation is an extension of intention preservation and attempts to account for issues with latency whereby a third collaborator may receive updates from two other clients out of order.

allow distributed, collaborative systems to mitigate this problem by transforming the specification of collaborators' updates given a history of how previous updates were applied.

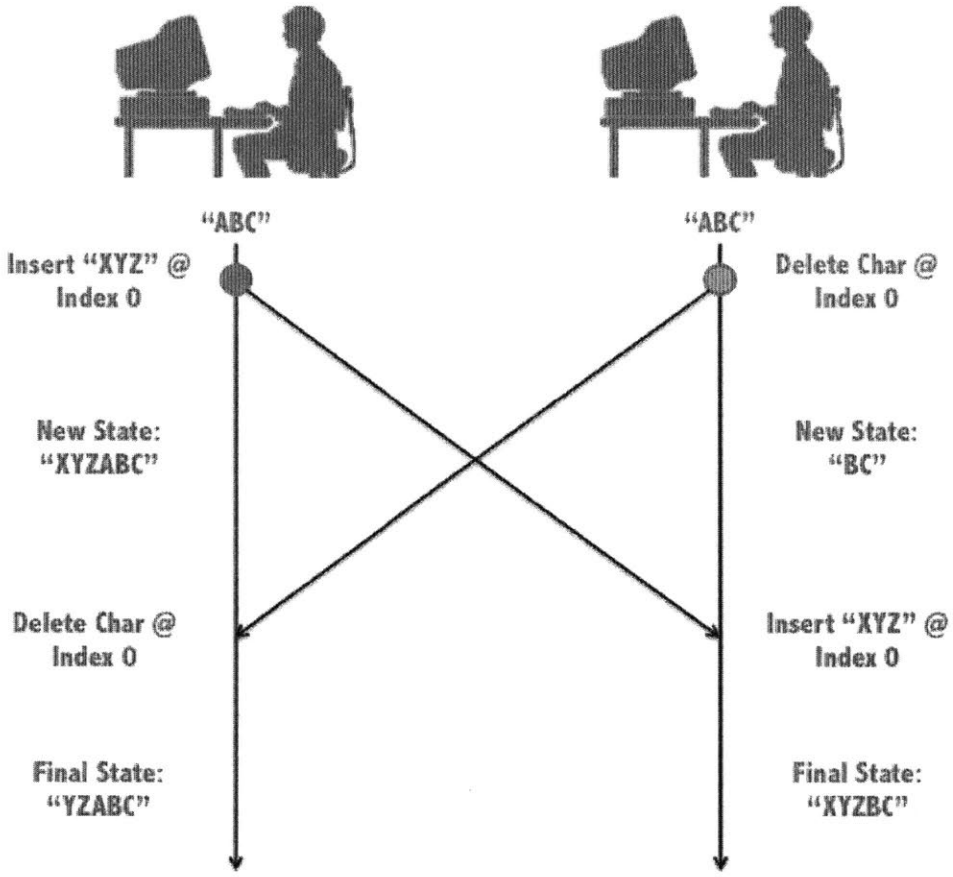


Figure 4-1: In this scenario, two editors try to change something at the same index, and catastrophe strikes as Blue intends to delete 'A' while Red ends up deleting 'X'.

What is striking about Red's problem? Well, first of all we notice that before the delete command, Red actually had all of the data he needed to end up in the right state! Upon analysis of the problem, the real issue was simply a miscommunication of what Blue intended. Now, can the miscommunication be ameliorated? Red knows that he added three characters, and as long as all clients had the same initial state, then Red could transform Blue's command to make sense in the context of his local update. How does Red do that? Well he simply considers his revision history and how the document has been changed since the initial string. From there, Red can

realize that from the original string, everything is now shifted over by three, and then transform Blue's command to preserve intention. In effect, this process is known as an operational transformation, and this example is specifically known as an inclusion transform, whereby a preceding operation has some effect on the current operation.

As much as school has taught us to harp on the incorrect, it is of equal merit to consider what ended up working correctly. In this example, we see that Blue actually ended up with the correct state. Why is that? As it turns out, since the deletion happened before the insertion at the same spot, the effect of the deletion could be excluded from the processing of the insertion. While this transformation was also an Inclusion Transform, we note that there was no net effect on the transform. This alludes to the complex logic that underlies these transforms, and shows that just because there was a previous update does not necessarily mean the operation needs to be transformed.

To illustrate the operational transformation process, consider Figure 4-2, whereby each collaborator analyzes and potentially transforms a given update. Before applying the update to the local state, the client figures out if it needs to transform a foreign operation to preserve intent. Ultimately, through this process, both clients are able to apply updates which preserve intention and converge to the same, correct state.

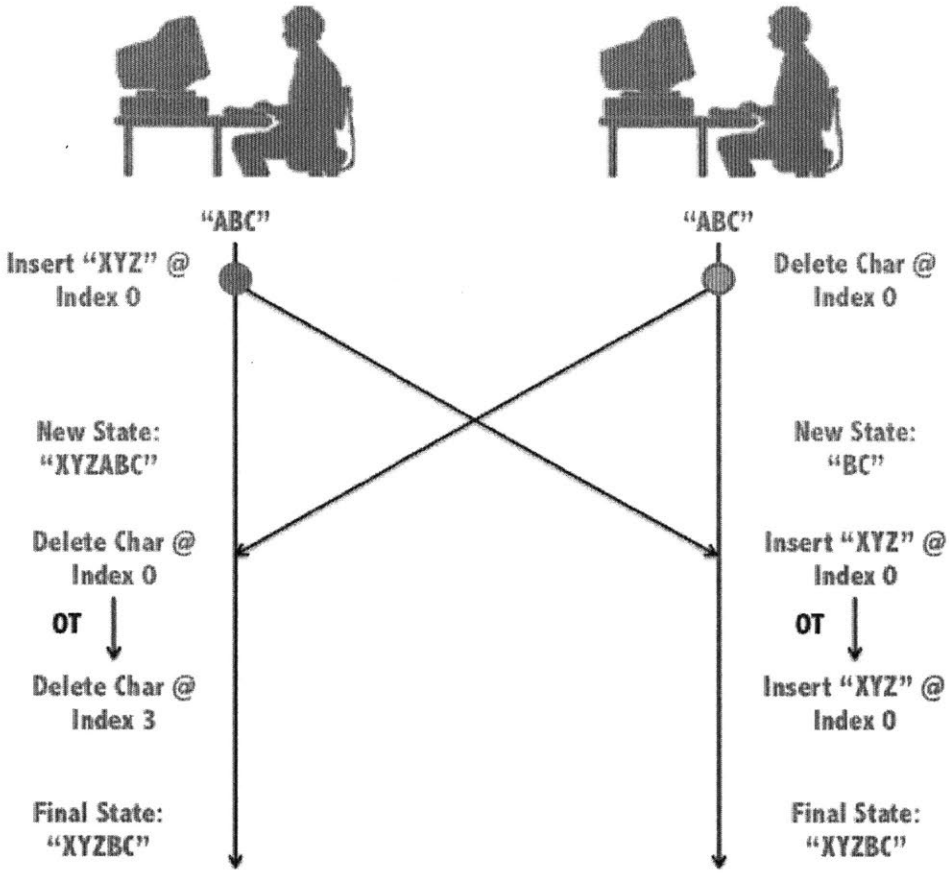


Figure 4-2: In this scenario, two editors try to change something at the same index, and the day is saved since Red resolves the intention of Blue's update through an Operational Transform.

4.2 Adapting Operational Transforms to JaDE

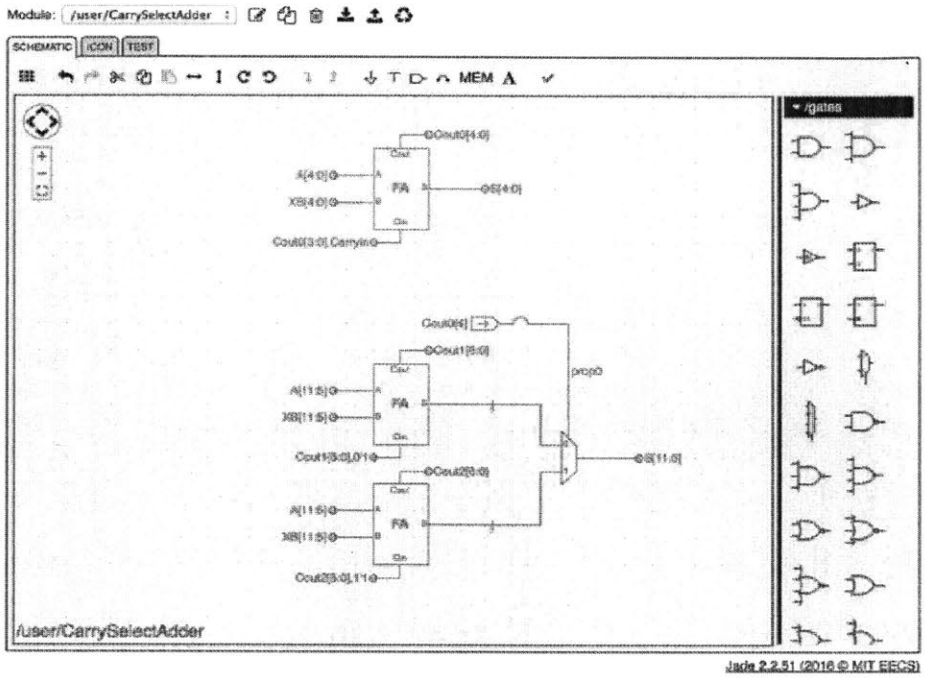
Now that the utility of operational transforms has been established, the next step is figuring out how to apply these string operations to JaDE. Luckily JaDE keeps all schematic information as an array of component data, which has the benefit of being linearly addressable, just like a string (Figure 4-3). Additionally, the way JaDE modifies and uses the schematics array is actually even simpler than a string. In JaDE, insertions can only happen to the end of the schematics array, and while deletion can happen anywhere in the schematic data structure, there is at most only one one component removed per atomic operation. The one added bit of complexity is that JaDE must support the ability to update the coordinates or the properties of a given circuit element, however, that simply requires rewriting an array at a given index, and has no effect on the indices of other components. When it comes to group operations, updates to individual components are applied iteratively, so the advantages of the atomic update hold, and will allow for simplification of the generalized operational transform algorithm. In summary, the types of operations needed to be supported by JaDE are as follows:

1. `Insert[Comp]` : Inserts a component into the schematic, by appending it to the end of the schematic array.

2. `Delete[Index]` : Removes the component at the specified index from the schematic array.

3. `Update[Comp]` : Finds the component in the schematic and overwrites its data.[3]

[3]Recall that the index is included in the persisted component data structure.




```
[Comp0, Comp1, ...]
```

```
['/lab3/FA',
[Coordinates],
{Properties},
0]
```

```
['/lab1/mux2',
[Coordinates],
{Properties},
1]
```

Figure 4-3: The data structure upon which all operational transforms will take place, either in the form of an insert, delete, or update.

4.3 Formalities of Operational Transforms

4.3.1 Definitions

Upon defining the problem specifically for JaDE, it is fruitful to examine operational transforms with some rigor, and to define the notation found in literature. Once the most crucial formalisms are established, a more traditional software approach is taken. To begin, we make the following definitions:

Definition 1 An Operation O_i is any of the aforementioned operations, with the subscript used for annotating certain operations.

Definition 2 A Causal Relationship $O_a \rightarrow O_b$ exists between two operations when they are produced sequentially by the same author. If causality is established, one may also say that O_b is dependent on operation O_a. For instance in Figure 4-2, the relationship between the two operations is not a causal relationship because they were not generated by the same author.

Definition 3 Independent Operations $O_a \mid\mid O_b$ exist when there is no causal relationship between the two operations. This occurs if the two operations were created by different authors. Therefore, in Figure 4-2, one would say that the two operations were independent.

Definition 4 Intention of Operation O_i is the effect that an operation would have had on the local state which generated it.

Definition 5 Execution Form of an Operation EO_i is the form of an operation O_i after an operational transform has been applied as to comply with intention preservation. In Figure 4-2, Red receives the operation "Delete Char @ Index 0" which is by way of operational transform converted to its execution form "Delete Char @ Index 3" before application to the local state.

Definition 6 Context CT_{O_a} is the state of the document in which operation O_a is generated. Going back to Figure 4-2, the context of the initial operations were both the string "ABC".

Definition 7 Context Equivalency Relation $O_a \sqcup O_b$ is a relationship that holds if the contexts of O_a and O_b are equivalent.

Definition 8 Context Preceding Relation $O_a \longmapsto O_b$ is a relationship that holds if and only if $CT_{O_b} = CT_{O_a} + [O_a]$. Context preceding relationships are non-transitive.

Definition 9 History Buffer HB is a local copy of every execution operation applied to the local document. This is required for undo/redo, and furthermore starting from the initial state, one can completely rebuild the state by applying the execution operations sequentially. In Figure 4-2, the final history buffer for Red would look like $HB =$ ["Insert 'XYZ' @ Index 0", "Delete Char @ Index 3"].

4.3.2 Fundamental Transforms

From these definitions, we can now completely specify the two fundamental types of transformations. The details of these transformations are lengthy and type-dependent, and their implementations can be found in Appendix E.

Inclusion Transformation $IT(O_a, O_b)$

1. Input parameters must be context equivalent i.e. $O_a \sqcup O_b$
2. Output is an operation EO_a which has the same effect on CT_{O_b} as O_a on CT_{O_a}.

Exclusion Transformation $ET(O_a, O_b)$

1. Input parameters must have the following precedence relation $O_b \longmapsto O_a$
2. Output is an operation $O_a{}'$ with $O_a{}' \sqcup O_b$ which has the same effect on $CT_{O_a{}'}$ as O_a on CT_{O_a}.

Formalism aside, an Inclusion Transform takes an operation O_b, and then updates another operation O_a to O'_a in order to account for the effect of O_b. The operation O'_a preserves the intent of O_a if it comes after an independent operation O_b.

On the other hand, an Exclusion Transform takes an operation O_b, and then updates another operation O_a to O'_a in order to undo the effect of O_b. The operation O'_a preserves the intent of O_a as if the independent operation O_b never happened.

4.3.3 Recursive List Transformations

With the definition of inclusion and exclusion transformations, we shall further define a function to apply a series of transformations on a given operation. As an added bonus, we can escape most of the mathematical formalism from here on out, so long as we heed what has been already established. For these two functions, it is important to remember the pre-conditions and post conditions as outlined in the previous section.

```
list_IT(op, list):
    if list.length == 0:
        return op
    else:
        list_IT(IT(op, list[0]), list[1:])


list_ET(op, list):
    if list.length == 0:
        return op
    else:
        list_ET(ET(op, list[0]), list[1:])
```

4.3.4 The Generalized Operational Transformation Algorithm

With the basics and formality established, it is time to define the algorithm in the terms laid out above. The Generalized Operational Transformation Algorithm takes two parameters, a newly received operation O_i (which may be from any collaborator, including yourself) and its full execution context for the local document, which is contained in the local History Buffer HB. The algorithm runs through the History Buffer and figures out how the operation needs to be changed. The algorithm returns the execution operation EO_i. Detailed examples for each case can be found in Appendix F. In writing, the algorithm breaks down into three distinct cases:

1. **Case 1:** *All operations in the History Buffer are causally preceding the operation O_i, and as a result you make no change to the operation.* In other words, you are the author of all changes in the document, and you can commit the update without performing a transformation.

2. **Case 2:** *Operations causally preceding O_i are listed before operations which are independent of the operation O_i. Therefore, we must transform O_i against these independent operations in sequence while ignoring the causally preceding operations.* In other words, there are other people who have edited the document and you need to account for their edits through the recursive inclusion transform.

3. **Case 3:** *Operations causally preceding O_i are jumbled up with operations which are independent of O_i. From here, we need to apply exclusion transforms to self generated updates to get the appropriate execution context for the operation O_i.* In this final case, which is very common, you have a mix of your own edits and others as well. Furthermore, the history buffer contains operations as they are executed on your document, which means that some of these operations will be double counted unless we exclude the independent operations from self-generated casual events. At this point, the exclusion transform is used to tease apart these instances of double counting and produce a correct execution operation EO_i for the current document context.

```
1 function got(op, history_buffer):
2
3   for i in range(length(history_buffer)):
4
5     // Case 2 detected
6     if independent(op, history_buffer[i]):
7
8       redundants = []
9       fixed_ops = []
10
11      for j in range(i, length(history_buffer)):
12
13        if causally_preceding(op, history_buffer[j]):
14
15          redundants.append(history_buffer[j])
16
17      if length(redundants) == 0:
18
19        return list_IT(op, history_buffer[i:])
20
21      else:
22        // Case 3 detected
23        causal_op = redundants[0]
24        index = causal_op.index
25
26        excluded_op = list_ET(causal_op, history_buffer[i:index].reverse())
27        fixed_ops.append(excluded_op);
28
29        for k in range(1, length(redundants)):
30
31          causal_op = redundants[k];
32          index = causal_op.index;
33
34          op_temp = list_ET(causal_op, history_buffer[i:index].reverse())
35          excluded_op = list_IT(op_temp, fixed_ops)
36          fixed_ops.append(excluded_op)
37
38        op_temp = list_ET(op, fixed_ops.reverse())
39
40        return list_IT(op_temp, history_buffer[i:])
41
42  // Default to Case 1
43  return op
```

Figure 4-4: Pseudocode for the Generalized Operational Transform algorithm.

4.4 Implementation

With the theory firmly established, we now look towards developing a real system which allows for collaboration between people. Using similar entry points to the modular update code, the collaborative form of JaDE of will now use Sockets instead of AJAX to update the data model. While this is different, the overall architecture of the network has not changed, with the server being a crucial point of interaction. Upon developing the Socket subsystem, a custom implementation of GOT was developed for JaDE, allowing for multiple users to now update a single JaDE document.

4.4.1 Network Architecture

For the network architecture of this distributed system, the traditional client-server model is still used. The reason for this architecture is two-fold. First, this allows much of the code to be reused and still preserve the centralized model critical for maintaining a coherent educational system. Secondly, it also reduces the number of connections that need to be managed. In a peer-to-peer network with n users, the system would have to support enough connections for the completely connected graph K_n, which scales as $O(n^2)$. Practically, this presents significantly more headaches than the client-server model, which needs to support connections for a star graph, which scales as $O(n)$. Additionally, since all updates pass through the server, it can ensure that causality is preserved for clients with flakier connections. In all, the traditional client model is vastly more appropriate for this project.

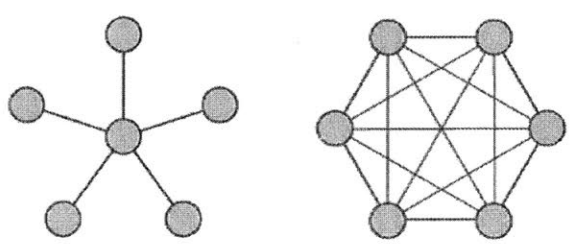


Figure 4-5: On the left, there is a graph representing the client-server model, and on the right there is a graph representing a peer-to-peer network. Since the network connections scale better and are more manageable in the client-server model, I continued with this architecture.

4.4.2 From AJAX to Sockets

With the selected client-server architecture and work from the previous chapter, we now just have to do a quick upgrade to retrofit the code with sockets as opposed to AJAX and replace the message handler class with an operational transformations class.

First and foremost, the socket library chosen to work with this system was Socket.io. Socket.io is a powerful and simple to use library which plugs nicely into the existing framework, and is already used in portions of the server. When originally considering ShareJS as an option, it was warned that Socket.io is non-causal and can result in problems and the library actively prevents you from importing Socket.io in projects. In an effort to all-around learn more for this thesis, I opted to throw caution to the wind, implement my own operational transform library, and then run it on a system using Socket.io.

Socket.io has many nice features, and beyond basic functionality the two of particular interest are namespaces and rooms. Namespaces allow for two separate domains to operate on the same server without interfering due to similarly named event handlers. The immediate application was in sequestering the queue[4] from the collaborative CAD schematics. As such, the server now has two namespaces, `schematics` and `queue`. Within a given namespace, clients can connect to a room, which is like a private subsection of the channel. In sharing documents, the host of the document will open a can define a room to which other participants connect and work collaboratively with.

[4]The queue is an existing Socket.io service used for running help queues in lab.

4.4.3 Design of the OT Class

Much like the previous chapter, a new separate class was designed in order to handle all modular communications, now over sockets instead of AJAX. With this new Operational Transform class, all of the OT mechanisms are self contained in the class and the class only exposes the ability to set an author identifier (which is used for insert-insert-tie breakers) and ability to broadcast an update to all collaborators. Inside the OT class, socket handlers are setup to listen for broadcast messages, which are then transformed via the GOT algorithm and then sent to the model to be incorporated into the model. Effectively, the OT class is providing a layer that sends out messages with their correct intent and provides incoming messages which have been modified by OTs to preserve the correct intent.

In order to make operational transforms work, additional metadata beyond what was supplied for modular updates must be added in order to preserve intention. In Figure 4-6, we see the OT update data structure has an added author field which gives a unique integer identifier to each participant in increasing connection order. The lower the number, the higher precedence your updates have which is critical for handling two inserts at the same location.[5] Additionally, there is an index field which is used to specify the location to be inserted in the schematic array data structure.

```
1 var ot_update = {
2   author: author_id_number,
3   update_type: update_type,
4   update_path: '/schematic/module:aspect',
5   update_data: Component,
6   index: index_to_be_in_schematics,
7   generation: generation_update_generated_on
8 };
```

Figure 4-6: Operational Transform Update data structure for use with sockets.

[5]The insert-insert tie is a conundrum that happens when you attempt to insert two elements into the same location. In order to ensure convergence, one must deterministically go before the other in the data structure. This issue is remedied by using a number ranking system with no two ranks being equal.

The final new field that is added is a generation field, which allows the operational transform system to tell when two transforms are independent and when they are causal. Recall that this is important for determining when to use inclusion and exclusion transforms, and the correct operation of the GOT algorithm depends on being able to find out the appropriate transform for use.

Using this data structure in conjunction with the GOT algorithm, I was able to create a simple OT-Enabled JaDE which was tested locally. The GOT algorithm was tested against a small number of handwritten test cases, so it is by no means extensively characterized, although it does seem to work. The source code for this system can be found in Appendix E, and it is written verbosely as to elucidate the underlying behavior of operational transforms.

Chapter 5

Discussion

As the this thesis comes to a close, it is important to reflect upon the work that has been done and the future work to be done. As such this chapter will cover the following topics:

1. Implications of Experimental Error
2. Future Work in Operational Transforms
3. Applications of Operational Transform Enabled CAD

5.1 Implications of Experimental Error

In Chapters Two and Three, efforts were made to reduce the overall bandwidth consumed by the JaDE application. As shown in Figures 2-6 and 3-11, the amount of bandwidth consumed by the application both on page load and in the process of circuit design were significantly reduced. In order to better understand these numbers, one should consider the experiment in its entirety. For these experiments, I went through the labs from start to finish once. The values presented in these figures are not averages but a mere single data point, however, I argue that is still a good metric of performance. As this is the umpteenth time I have gone through the labs, I was able to pass all tests with relatively few problems, with improper bus widths generally being the culprit. That being said, I still made far fewer updates than the average user for these labs, and as a result I believe to have produced a viable lower bound on the data transferred for each of these labs. Therefore, while I doubt the numbers will line up to mine exactly in practice, the absolute total bandwidth saved is likely to be far greater in practice that what I reported.

Additionally, one should question whether or not my results captured the actual trend of bandwidth savings. I personally doubt that the curves show the correct trend except for the massive savings. What I would expect in practice is to see Lab 3 total bandwidths to be much higher than Lab 5 total bandwidths. I figure this will be the case as Lab 3 it is a much more difficult lab than Lab 1, and students are still learning how to use JaDE. By the time students need to design the Beta, they will have been acquainted with all features of JaDE that they need to succeed and make fewer mistakes due to the learning curve of the tool.

In summary, while I believe that the data I took in Chapters Two and Three shows that the features I implemented will greatly reduce bandwidth, I foresee any collected data to show both greater absolute savings per student as well as a slightly different trend in bandwidth consumption.

5.2 Future Work in Operational Transforms

One of the most interesting facets of operational transforms is the fact that many different algorithms and schemes exist in order to accomplish the same goal. As further research in this area, it would be prudent to implement and test various other algorithms to see which works best in practice and which confer the most features with the least headache. I would argue that actual implementation and testing is necessary, as history has shown that several theoretically better algorithms (lower time or space complexity) end up performing worse than their more naive counterparts (see comparisons of binary heaps and fibonacci heaps).

For future work, it would be advisable to implement the optimized version of the Generalized Operational Transform Algorithm. This is fairly straightforward, but I have simply run out of time. Additional algorithms worth implementing and testing are Jupiter OT and Google Wave OT. As a note on the implementation, it is worth considering that components are added to the JaDE environment in an atomic fashion analogous to character operations on a string. In addition to the traditional straightforward operational transform, implementing a scheme to support undo/redo would be very interesting and worthwhile. As it stands, undo/redo requires teasing apart your own schematic updates from other people's, and then somehow only undoing what you did. This is not an easy task, and is truly required for a fully functional CAD environment.

5.3 Applications of OT-Enabled CAD

Now that there is a JaDE platform capable of Operational Transforms, the next places to look are the applications of OT-Enabled. The first step would be to fully revamp the 6.004 server to support socket-based updates instead of AJAX calls, using the testbed as a template. Upon developing this platform, the namespaces in conjunction with socket rooms (as explained in the previous chapter) allows for private communication over a namespace among a few trusted people. Therefore, these socket rooms can mediate communication for a collaborative editor. Upon connecting to a mongo database and using the OT-Enabled JaDE, it will be possible to construct at least two novel technologies which may prove useful in education.

5.3.1 JaDE Hangouts

In order to facilitate remote communication between course staff and students, as well as between students, a potential new technology could be OT-Enabled JaDE embedded in a Google Hangout. This would allow students from across campus (or even the world!) to actively engage with students and/or other staff while working together to design digital circuits. Having direct communication through a Skype-like call or even an on-screen chat window could prove to be quite useful.

5.3.2 Collaborative Classroom

Another area of interest would be in developing a collaborative classroom with a public workbook. In this model, a course instructor could design a circuit on an overhead display, and ask questions. As the instructor is asking questions, collaboration could be enabled so students can very easy change the schematic and ask related questions such as (what happens if we place an AND gate here) or collaboratively design lenient gates and run simulations to get immediate feedback and see results. I envision this would make certain lessons much more active and hands on.

Appendix A

Schema References

In this appendix are diagrams of the databases and data structures that were interacted with over the course of this thesis. The hope is that this appendix will serve as a clean, quick reference guide for those wishing to interact with the data structures for either code maintenance or future research work.

Key Terms

1. **Module:** A Module is a data structure which contains all of the information for a given circuit block, comprised of other data structures referred to as Aspects. The four types of Aspects are Icon, Properties, Schematic, and Test.

2. **Icon Aspect:** An aspect defining what the icon for the device should look like.

3. **Properties Aspect:** Miscellaneous metadata for the circuit block.

4. **Schematic Aspect:** Array of wires and gates, with each having coordinates and connections specified.

5. **Test Aspect:** A text block which contains the test vectors for circuit schematic.

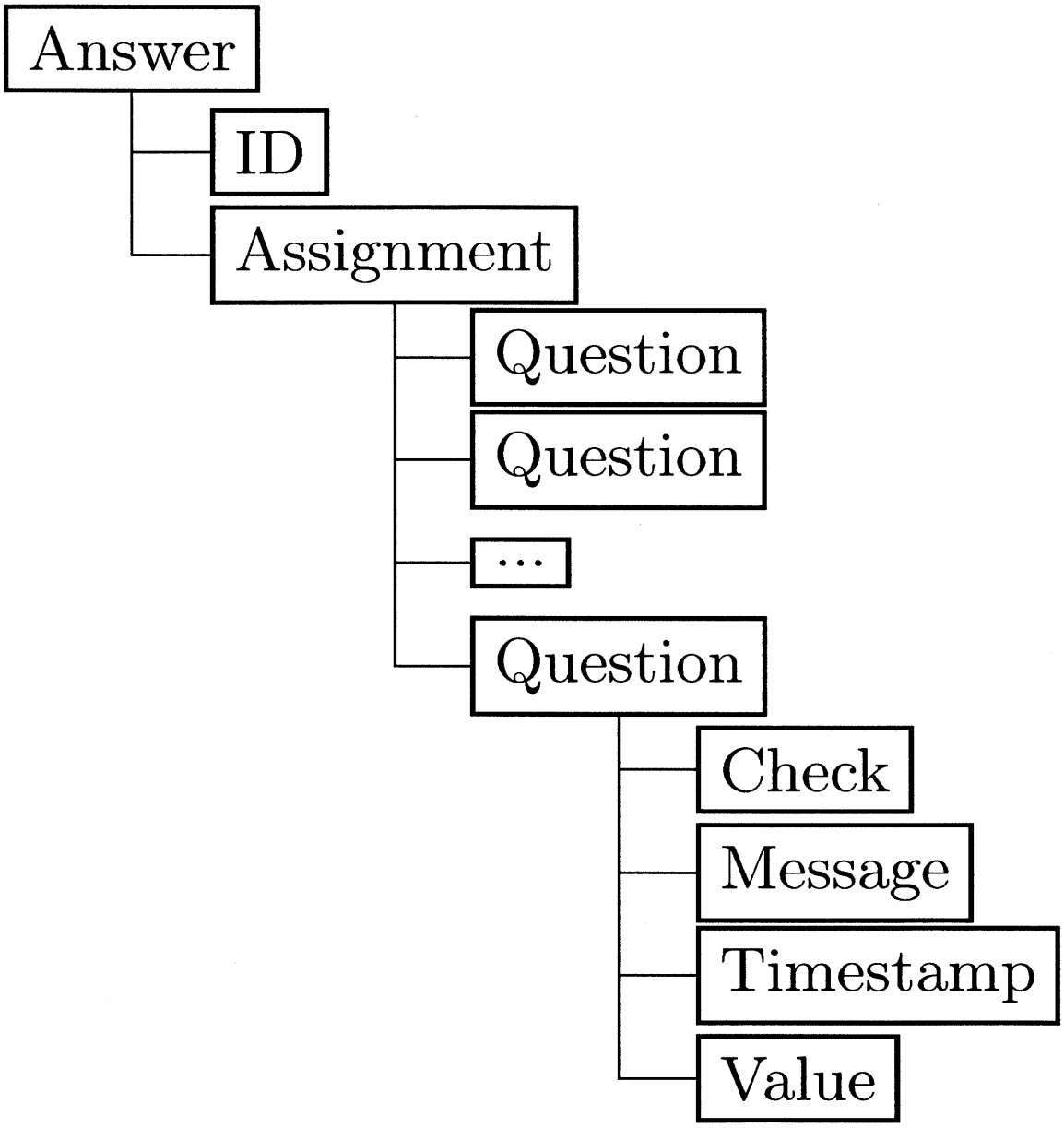


Figure A-1: A collection of these constitute the members of the Answers Database. The Answers Database is used by the server backend to persist both lab materials (i.e. the data JaDE uses) and answers to other questions.

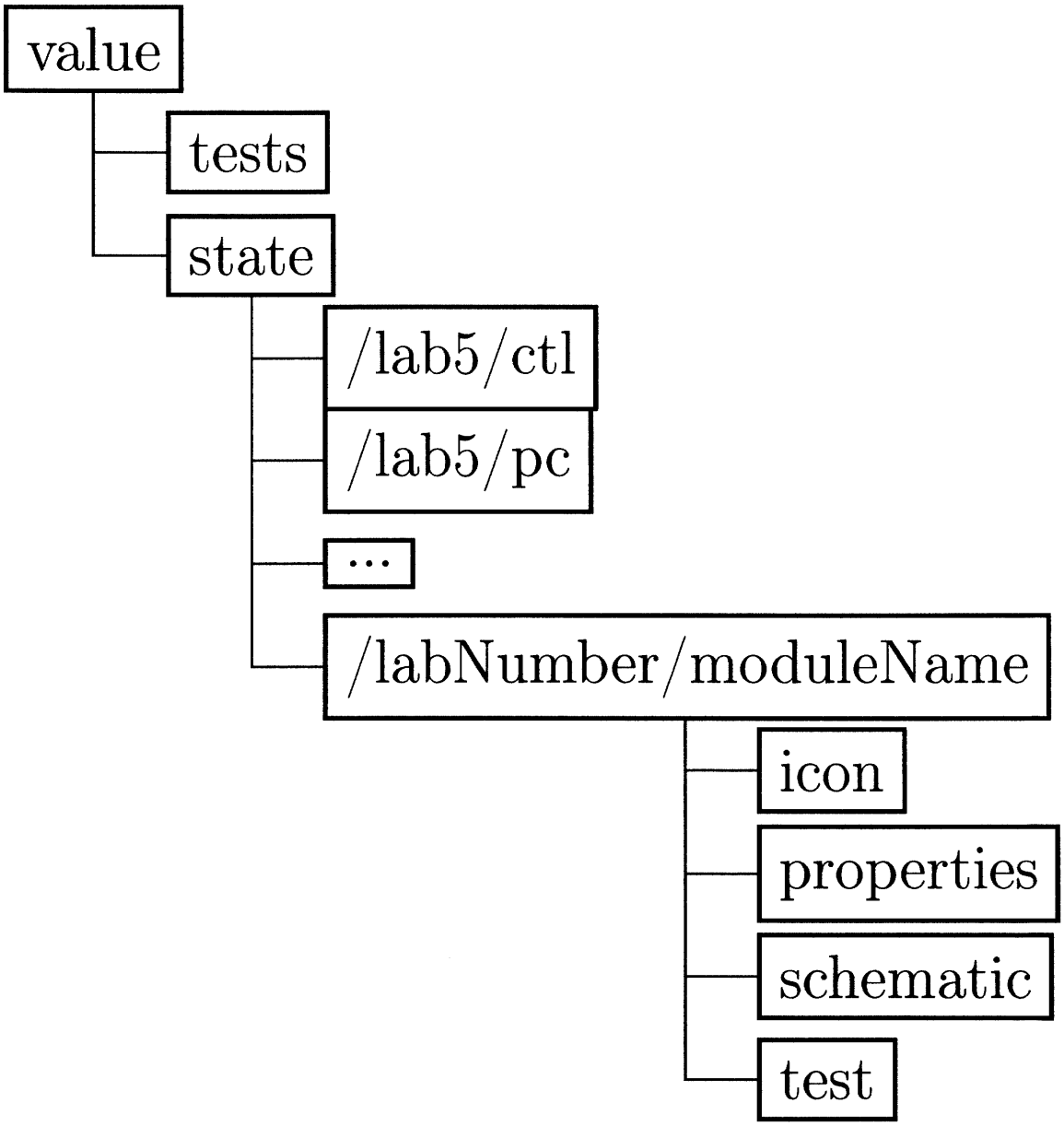


Figure A-2: Note that within a given lab, there are several different modules, each of which has a State containing the four Aspects of Icon, Property, Schematic, and Test. The Tests field which is a direct child of Value stores whether or not the State's Modules have passed their tests.

Appendix B

Script Listings

In this appendix, there are the code listings of all scripts involved in this thesis. These pieces of code do not have anything to do at runtime of the application, but nonetheless played an integral role in establishing the infrastructure for the updated server. Scripts that are listed herein are pertinent to the material covered in:

1. Hash Build Process (Central Repository Creation and HTML Compilation)
2. Answers Database Reformatting

```
1 ## HTML Parser and Hash Replacement Strategy
2 ## Jonathan Terry
3 import glob,json,os.path,hashlib
4 from BeautifulSoup import BeautifulSoup
5
6 # Hack to make the UTF-8 play nicely
7 import sys
8 reload(sys)
9 sys.setdefaultencoding('utf-8')
10
11 # File name of the lab repo
12 __repo__ = "repo.json"
13
14 def compile_md5():
15
16   # Read the repo with the master aspects and initialize
17   # a Python dictionary and a json dump to serve from.
18   initial_state_string = ""
19   initial_state_file = {}
20
21   with open(__repo__, 'r') as file:
22     for line in file:
23       data = line.strip()
24       initial_state_string = initial_state_string + data
25
26   initial_state_hashes = json.loads(initial_state_string)
27
28   for module in initial_state_hashes.keys():
29     for aspect in initial_state_hashes[module]:
30       aspect_string = json.dumps(initial_state_hashes[module][aspect])
31       aspect_hash = hashlib.md5(aspect_string).hexdigest()
32       initial_state_hashes[str(module)][str(aspect)] = aspect_hash
33       initial_state_file[aspect_hash] = aspect_string
34
35   with open('_initial_state.json','w') as f:
36     json.dump(initial_state_file,f)
```

```
37  # Iterate through each lab template, replacing
38  # bulky JSON as needed in the answer tags
39
40  for fname in glob.glob('*/_lab.html'):
41
42    lab = fname.split('/')[0]
43
44    f = open(fname)
45    contents = f.read()
46    f.close()
47
48    # In lieu of an illegible regex, I opted for using
49    # BeautifulSoup in order to make the program easier to
50    # maintain and easier to understand.
51    # To get BeautifulSoup, run "sudo easy_install BeautifulSoup"
52
53    soup = BeautifulSoup(contents);
54    schematics = soup.findAll('answer', {'tool_name' : 'Jade'})
55
56    for schematic in schematics:
57
58      initial_state_md5 = {}
59
60      try:
61        template =  schematic.text
62        json_mapping = json.loads(str(template))
63
64        for module in json_mapping['initial_state']:
65
66          mod = {}
67
68          for aspect in json_mapping['initial_state'][module]:
69
70            mod[aspect + '.hash'] = initial_state_hashes[module][aspect]
71
72          initial_state_md5[module] = mod
```

```
73          if 'initial_state' in json_mapping:
74
75            del json_mapping['initial_state']
76            json_mapping['initial_state_md5'] = initial_state_md5
77
78            replacement_data = schematic
79            replacement_data.string = json.dumps(json_mapping)
80
81            html = soup.prettify("utf-8")
82
83            with open(os.path.join(lab,'lab.html'), 'w') as file:
84                file.write(html)
85
86        except (AttributeError, KeyError):
87          pass
```

Figure B-1: Code listing for MD5 build process that remakes the labs given a central repository of master Aspects.

```
1  // Reformat.js
2  // Script by Jonathan Terry
3  // Iterates through the answers collection, changing the
       stringied fields to be that of a JSON object
4
5  // First lets get into the Spring 16 database
6  use S16
7
8  // Extract all of the files into a dummy collection
9  db.answers.copyTo('dummyCollection')
10
11 // Extract the stringified JSON and then repackage it as JSON
12 db.answers.find().forEach( function(answer) {
13
14     var current_id;
15
16     for (key in answer) {
17
18       try {
19
20         // Only go deeper into particular assignments
21         if (key != '_id') {
22
23           var assignmentLevel = answer[key];
24
25           // Iterate over the assignments.
26           for (question in assignmentLevel) {
27
28             var questionContent = assignmentLevel[question];
29
30             for (item in questionContent) {
31
32               var check = questionContent[item];
33
34
35               if (check[0] === '{') {
```

```
36
37                  // Convert to JSON, and then update the entry,
                       deriving the keys
38                  var labContent = JSON.parse(check);
39                  var pathToLab = key + '.' +question + '.' + item;
40                  var obj = {};
41
42                  obj[key] = assignmentLevel;
43                  obj[key][question][item] = labContent;
44
45                  db.dummyCollection.update({ '_id' : current_id },
                        { $set:  obj });
46                }
47              }
48            }
49          }
50          else {
51
52            var current_id = answer._id;
53          }
54
55        }
56
57        catch (e) {
58          print(e);
59          continue;
60        };
61      };
62    }
63  )
64
65  db.dummyCollection.copyTo('answersX');
```

Figure B-2: Code listing for parsing existing Answers mongo databases. Note that this requires changes to the model (i.e. removal of JSON.parse) handling in JaDE. Should execute without throwing an error.

Appendix C

Caching Mechanism Code

In this appendix, the JaDE code relevant to the MD5 handshake caching system is listed, in the form of a single cohesive javascript file which can be pulled in through the build process and add functionality to any javascript build. Existing code written by others lacks the javadoc comments. The following functions are contained herein.

1. `jade.client_server_sync(md5_config)`
2. `request_initial_state(missing_aspects, callback)`
3. `jade.collect_trash(hashes)`
4. `jade.construct_initial_state(md5_config)`
5. `jade.detect_storage(type)`

```
1  /**
2   * Synchronizes read-only material in the localStorage with the
      server's most recent files
3   * @param  {JSON} md5_config - Array of MD5 Hashes that the
      server expects the client to have
4   */
5  jade.server_client_sync = function(md5_config) {
6
7    if (!this.detect_storage) {
8      throw new Error("Unable to invoke localStorage, please use an
          HTML5 compatible browser.");
9    }
10
11   var missing_aspects = [];
12   var intended_hashes = {};
13   for (module in md5_config) {
14     var module_object = md5_config[module];
15     for (aspect in module_object) {
16
17       var hash = module_object[aspect];
18       intended_hashes[hash] = true;
19
20       if (localStorage.getItem(hash) === null) {
21         missing_aspects.push(hash);
22       };
23     };
24   };
25
26   if (missing_aspects.length > 0) {
27     this.request_initial_state(missing_aspects,
          this.update_local_storage);
28   };
29
30   this.collect_trash(intended_hashes);
31   return this.construct_initial_state(md5_config);
32 };
```

```
33 /**
34  * Helper function to coordinate data exchange with the server.
35  * @param  {Array}   missing_aspects - Array of the aspect MD5
      hashes to be updated.
36  * @param  {Function} callback       - Function to handle the
      JSON response from the server.
37  * @throws {Error} If localStorage not supported in browser.
38  */
39 jade.request_initial_state = function(missing_aspects, callback)
     {
40
41   for (var i = 0; i < missing_aspects.length; i++) {
42     var hash = missing_aspects[i];
43     var args = {
44       async: false, // Allow the schematics to come in chunks
45       url: '/user/initial_state/' + hash,
46       type: 'GET',
47       datatype: 'json',
48       error: function(jqXHR, textStatus, errorThrown) {
49         alert('Error while loading library '+filename+': '+
             errorThrown);
50       },
51       success: function(json) {
52         if (callback) callback(hash, json);
53       }
54     };
55     $.ajax(args);
56   };
57 };
58
59
60
61
62
63
64
```

```
65 /**
66  * Function to persist data
67  * @param  {String} hash   - MD5 hash corresponding to an Aspect
68  * @param  {String} aspect - Stringified data structure
       representing the
69  */
70 jade.update_local_storage = function(hash, json) {
71   var data = json[hash];
72   localStorage.setItem(String(hash), String(data));
73 };
74
75 /**
76  * Gets rid of unnecessarily cached items
77  * @param  {JSON} hashes JSON mapping with MD5 hashes as keys
78  */
79 jade.collect_trash = function(hashes) {
80
81   console.log(hashes);
82
83   for (var i = 0; i < localStorage.length; i++) {
84
85     var key = localStorage.key(i);
86
87     if (key in hashes) {s
88       continue;
89     }
90
91     else {
92       localStorage.removeItem(String(key));
93     };
94
95   };
96
97 };
98
99
```

```
100 /**
101  * Glues together the intitial_state specification and the
        associated cached data
102  * @param  {JSON} md5_config - Mapping of Aspect names to their
        MD5 Hashes
103  * @return {JSON} init_state - Mapping of Modules to their data
104  */
105 jade.construct_initial_state = function(md5_config) {
106
107   var init_state = {};
108
109   for (module in md5_config) {
110
111     var compiled_module = {};
112     var module_object = md5_config[module];
113
114     for (key in module_object) {
115
116       var aspect = String(key).split('.')[0]
117       compiled_module[aspect] = JSON.parse(localStorage.getItem(
            module_object[key]));
118
119     };
120
121     init_state[module] = compiled_module;
122
123   };
124
125   return init_state;
126
127 };
128
129
130
131
132
```

```
/**
 * Detects whether or not the browser supports HTML5 storage
 * @param  {String}     type - Either 'sesstionStorage' or '
    localStorage' depending on what you are testing
 * @return {Boolean}    bool - Whether or not the type of storage
     requested is actually available
 */
jade.detect_storage = function(type) {

  try {
    var storage = window['localStorage'];
    var test = '__STORAGE_TEST__';

    storage.setItem(test, test);
    storage.removeItem(test);

    return true;
  }
  catch (e) {
    return false;
  };
};
```

Figure C-1: Code listed here is used for the client side caching and MD5 handshake procedures. These listings are quite awesome.

Appendix D

Modular Updates Code

Listed herein is the bulk of the new code that I introduced to Jade and the server in order to allow for modular updates. This appendix lists the following upgrades to classes:

1. `Jade`
2. `Component`
3. `Message Handler`
4. `Server Upgrades`

```
1  // Similar to above, but takes the tip as a function
2  Jade.prototype.module_tool_dynamic = function (icon,id,tip,action
       ,extra_classes) {
3      var tool = $('<span></span>').append(icon).addClass('
           jade-module-tool jade-tool-disabled').attr('id',id);
4      if (extra_classes) tool.addClass(extra_classes);
5
6      var j = this;  // for closure
7      tool.on('click',function (event) {
8          if (action) action(j,event);
9          event.preventDefault();
10         return false;
11     });
12
13     tool.on('mouseenter',function () {
14         j.status.html(tip());
15     });
16
17     tool.on('mouseleave',function () {
18         j.status.html('');
19     });
20
21     return tool;
22 };
```

Figure D-1: Upgrade to the Jade class allowing for a more dynamic GUI.

```
1  Component.prototype.add = function(aspect) {
2      this.aspect = aspect; // we now belong to a diagram!
3      this.index = this.aspect.components.length;
4      aspect.add_component(this);
5      this.update_coords();
6
7      // create a record of the change
8      var component = this; // for closure
9      function component_add(diagram, action) {
10         if (action == 'undo') component.remove();
11         else component.add(diagram);
12     }
13
14     aspect.add_change(component_add);
15
16     if (jade.model.get_init_complete()) {
17
18         var path = this.get_path();
19         var component_data = this.get_data_model();
20
21         jade.model.send_component_update("INSERT", path,
               component_data);
22
23     };
24 };
```

Figure D-2: Upgrade to the Component Class which allowed for modular insert.

```
1  Component.prototype.remove = function() {
2      // remove connection points from diagram
3      for (var i = this.connections.length - 1; i >= 0; i -= 1) {
4          var cp = this.connections[i];
5          this.aspect.remove_connection_point(cp, cp.location);
6      }
7
8      // remove component from diagram
9      this.aspect.remove_component(this);
10
11     // create a record of the change
12     var component = this; // for closure
13     function component_remove(diagram, action) {
14         if (action == 'undo') component.add(diagram);
15         else component.remove();
16     }
17
18     this.aspect.add_change(component_remove);
19
20     if (jade.model.get_init_complete()) {
21
22         var path = this.get_path();
23         var component_data = this.get_data_model();
24
25         jade.model.send_component_update("DELETE", path,
               component_data);
26
27         this.aspect.refactor_components();
28     };
29 };
```

Figure D-3: Upgrade to the Component Class which allowed for modular delete.

```
1  Component.prototype.update_properties = function(new_properties)
     {
2      if (new_properties !== undefined) {
3          var old_properties = this.clone_properties(false);
4          this.properties = new_properties;
5          this.compute_bbox();
6
7          var component = this; // for closure
8          function component_update_properties(diagram, action) {
9              if (action == 'undo') component.properties =
                   old_properties;
10             else component.properties = new_properties;
11             this.compute_bbox;
12         }
13
14         this.aspect.add_change(component_update_properties);
15
16         if (jade.model.get_init_complete()) {
17
18             try {
19
20                 var path = this.get_path();
21                 var component_data = this.get_data_model();
22
23                 jade.model.send_component_update("UPDATE", path,
                       component_data);
24
25             } catch(e) {
26
27             };
28         };
29     }
30 };
```

Figure D-4: Upgrade to the Component Class which allowed for modular update.

```
1  Component.prototype.get_data_model = function() {
2
3      var data_model;
4
5      // Check for gates, since their properties are not persisted.
6      if (this.type()[0] === '/' || this.properties == {}) {
7
8          data_model = [this.type(), this.coords, this.index];
9
10     } else {
11
12         data_model = [this.type(), this.coords, this.properties,
               this.index];
13
14     };
15
16     return data_model;
17 };
18
19 Component.prototype.get_path = function() {
20     return this.aspect.module.name + ':' + this.aspect.name;
21 };
```

Figure D-5: Upgrade to the Component Class which extracted the Mongo data model and target path.

```
1
2 ////////////////////////////////////////
3 // Messaging and State Synchronization //
4 ////////////////////////////////////////
5
6
7 // Server Request Data Structure
8 //
9 // var server_req = {
10 //
11 //   id: request_number,
12 //   gen: generation_number
13 //   updates: list_of_updates
14 //
15 // }
16
17 jade_defs.message_handler = function(jade) {
18
19   // FIFO to maintain any backlogged updates
20   var message_queue = [];
21
22   // For state coherence with server
23   var generation = undefined;
24
25   // Not currently used, but here just in case :)
26   var timestamp = undefined;
27
28   // Tip information
29   var tip = 'Connected to server and synced.'
30
31   // Prevents annoying popups
32   var allow_alerts = true;
33
34   function get_queue_generation() {
35     return generation;
36   }
```

```
37
38  function get_message_queue() {
39    return message_queue;
40  }
41
42  function get_tip() {
43    return tip;
44  }
45
46  /**
47   * Enqueues the request for the server, triggering a push if a
       limit is reached
48   * @param  JSON update - Server Request JSON object
49   */
50  function enqueue_request(update) {
51
52    // Graphics update to indicate changes
53    set_pending();
54
55    if (update.update_type === "RESET") {
56
57      hard_reset(update);
58
59    }
60    else {
61
62      message_queue.push(update);
63
64      if (message_queue.length >= jade.model.AUTOSAVE_TRIGGER) {
65
66        push_changes();
67
68      };
69    };
70
71  };
```

```
72
73  /**
74   * Clears the messages from the message queue, and pushes any
        pending updates
75     * to the message queue, and possibly the server if the
          autosave trigger
76   * is surpassed. Also updates the generation number.
77   */
78  function clear_message_queue() {
79
80    message_queue = [];
81    generation += 1;
82
83    if (message_queue.length >= jade.model.AUTOSAVE_TRIGGER) {
84      push_changes();
85    }
86    else {
87      set_synced();
88    };
89
90  };
91
92
93  /**
94   * Clears all of the queues and sends the reset command to the
        server.
95   * @param  JSON update Updates containing the reset command.
96   */
97  function hard_reset(update) {
98
99    message_queue = [update];
100   push_changes();
101
102 };
103
104
```

```
105   /**
106    * Pushes all pending changes to the server.
107    */
108   function push_changes() {
109
110     // Prevents against sending nothing
111     if (message_queue.length == 0) {
112       return;
113     }
114
115     var server_req = {
116
117       generation: generation,
118       updates: JSON.stringify(message_queue)
119
120     };
121
122     $.ajax({
123       async: false,
124       url: '/user/lab' + jade.model.get_mongo_path(),
125       type: 'POST',
126       datatype: 'json',
127       data: server_req,
128       success: function(json) {
129         allow_alerts = true;
130         timestamp = json.timestamp;
131         clear_message_queue();
132       },
133       error: function() {
134         server_connection_error();
135       }
136     });
137   };
138
139
140   /**
```

```
141     * Checks to see if the server has a newer version of the
          schematic,
142     * and if so it alerts the user to the difference. For use when
          laptop is shut.
143     */
144    function compare_states() {
145
146      var stale_generation = generation;
147
148      check_connection();
149
150      if (stale_generation < generation) {
151        inconsistent_state_alert();
152      }
153
154    }
155
156    /**
157     * Called on initial load, used to get a coherent generation
          number as well as
158     * alert the user to multiple instances of Jade being open.
159     */
160    function establish_connection() {
161
162      if (jade.model.get_init_complete()) {
163
164        $.ajax({
165          async: false,
166          url: '/user/lab/sync' + jade.model.get_mongo_path() + '/
             open',
167          type: 'GET',
168          datatype: 'json',
169
170
171
172
```

```
173         success: function(json) {
174
175                 generation = parseInt(json.generation);
176
177                 if (parseInt(json.open_jade) > 1) {
178                   multiple_instance_error();
179                 }
180             },
181         error: function() {
182           server_connection_error();
183         }
184       });
185     }
186
187   };
188
189   /**
190    * Proper way to disconnect from a Jade session.
191    */
192   function teardown_connection() {
193
194     $.ajax({
195       async: false,
196       url: '/user/lab/sync' + jade.model.get_mongo_path() + '/
          close',
197       type: 'GET',
198       datatype: 'json',
199       error: function() {
200         server_connection_error();
201       }
202     });
203
204   };
205
206
207
```

```
208   /**
209    * Can be used in the event that the open jade count gets
          screwed up due to
210    * improper disconnects. Manual use only.
211    */
212   function reset_connection() {
213
214     $.ajax({
215       async: false,
216       url: '/user/lab/sync' + jade.model.get_mongo_path() + '/
            reset',
217       type: 'GET',
218       datatype: 'json',
219       error: function() {
220         server_connection_error();
221       }
222     });
223   };
224
225   /**
226    * Checks to see if the server is on a different generation.
227    */
228   function check_connection() {
229
230     $.ajax({
231       async: false,
232       url: '/user/lab/sync' + jade.model.get_mongo_path() + '/
            check',
233       type: 'GET',
234       datatype: 'json',
235       success: function(json) {
236
237                 generation = parseInt(json.generation);
238
239                 if (parseInt(json.open_jade) > 1) {
240                   multiple_instance_error();
```

```
241                 }
242             },
243       error: function() {
244         server_connection_error();
245       }
246     });
247
248   };
249
250   /**
251    * Contacts the server for an entire copy of the state. Use
         sparingly.
252    */
253   function get_complete_state() {
254
255     var lab = jade.model.get_mongo_path().split('/')[0];
256     var id = jade.model.get_mongo_path().split('/')[1];
257
258     $.ajax({
259       async: false,
260       url: '/user/lab/' + lab,
261       type: 'GET',
262       datatype: 'json',
263       success: function (json) {
264         // Delete our garbage
265         delete jade.configuration.state;
266         // Reinitialize jade
267         jade.initialize(json.lab.id.value.state);
268         set_synced();
269       },
270       error: function() {
271         server_connection_error();
272       }
273     });
274
275   };
```

```
276   ////////////////////
277   // Error Messages //
278   ////////////////////
279
280   function server_connection_error() {
281
282     var title = "SERVER CONNECTION ERROR";
283     var content = $('<div style="margin:10px;width:300px;">Unable
          to contact server. Another attempt will be made
         automatically in 5 minutes<span id="mname"></span>. Press
         okay to attempt an update now.</div>');
284     var offset = $('.jade-tabs-div',jade.top_level).offset();
285
286     set_error();
287
288     function call_me() {
289       push_changes();
290     }
291
292     if (allow_alerts) {
293       allow_alerts = false;
294       jade.dialog(title, content, call_me, offset);
295     }
296
297
298   }
299
300   function multiple_instance_error() {
301
302     var title = "JADE OPEN ON ANOTHER COMPUTER";
303     var content = $('<div style="margin:10px;width:300px;">It
         appears as if another instance of Jade is running. This is
          dangerous and can result in lost or corrupted data<span
         id="mname"></span>. Press okay to close window and return
         to safety.</div>');
304     var offset = $('.jade-tabs-div',jade.top_level).offset();
```

```
305
306     set_error();
307
308     function call_me() {
309       window.close()
310     }
311
312     jade.dialog(title, content, call_me, offset);
313
314   }
315
316   function inconsistent_state_alert() {
317
318     var title = "STATE HAS CHANGED SINCE YOU WERE LAST ONLINE";
319     var content = $('<div style="margin:10px;width:300px;">The
          data on the server is newer than your current state<span
          id="mname"></span>. Press okay to get the server data.
          </div>');
320     var offset = $('.jade-tabs-div',jade.top_level).offset();
321
322     // Regardless of what is clicked, we get the entire new state
           from the server
323     get_complete_state();
324
325     function call_me() {
326
327     }
328
329     jade.dialog(title, content, call_me, offset);
330
331   };
332
333
334
335
336
```

```
/**
 * Updates status.
 */
function set_synced() {


  $("span.connection_status_indicator").find("span").text("Synced!").css("color", "black");
  $("span.connection_status_indicator").removeClass("jade-tool-enabled").addClass("jade-tool-disabled");
  tip = 'Connected to server and synced.';

};

function set_pending() {

  $("span.connection_status_indicator").find("span").text("Changes Pending...").css("color", "darkgoldenrod");
  $("span.connection_status_indicator").removeClass("jade-tool-disabled").addClass("jade-tool-enabled");
  tip = "Connected to server, but changes not saved. Click to save changes.";

};

function set_error() {
  $("span.connection_status_indicator").find("span").text("Connection Error!").css("color", "red");
  $("span.connection_status_indicator").removeClass("jade-tool-disabled").addClass("jade-tool-enabled");
  tip = "Errors with synchronization. Click to resolve.";
}

// Helper functions to maintain state in the event of closing the laptop,
// browser, or even the ~dreaded~ double editing.
```

```
365
366   // For x-ing out browser window
367   $(window).bind('beforeunload',function () {
368
369     // Only do this if we care about saving the data
370     if (jade.model.get_mongo_path()){
371       jade.save_to_server();
372       push_changes();
373       teardown_connection();
374     }
375
376       return undefined;
377   });
378
379   // Covers your bases if idle (pushes every 10 seconds)
380   setInterval(function() { push_changes() }, 10000);
381
382   // Checks for laptop shut
383   var previous = (new Date).getTime();
384   setInterval(function() {
385
386     var current = (new Date()).getTime();
387         if (current-previous > 3000) {
388             compare_states();
389         }
390         previous = current;
391
392   }, 1000);
393
394
395
396
397
398
399
400
```

```
401     ////////////////////
402     // Module Exports //
403     ////////////////////
404
405     return {
406
407       enqueue_request: enqueue_request,
408       push_changes: push_changes,
409       establish_connection: establish_connection,
410       get_tip: get_tip
411
412     };
413
414   };
```

Figure D-6: Message Handler Class Entire Listing.

```
1  // Route handling modular updates to the underlying data model
2  // Schematic aspects are the only thing that is updated at the
       Component Level
3  // All other Aspects are updated in their entirety due to their
       smaller size
4  // The request body contains all of the data required for an
       update
5  // Existing machinery is in place to transmit the entire model
       need be
6  router.post('/lab/:lab/:problem/', function(req, res, next) {
7
8      // Used to point to the home of our component we are updating
9      var kerberos = req.session.user;
10     var lab = req.params.lab;
11     var problem = req.params.problem;
12
13     var lookup = {};
14     var value = "value";
15     var state = "state";
16
17     // Get the generation of the update as well as the data
18     var generation = req.body.generation;
19     var list_of_updates = JSON.parse(req.body.updates);
20
21     lookup[lab] = 1;
22     req.local.answersX.findOne({ _id: kerberos}, lookup, function
           (err, answer) {
23
24         // Extract the module that we are interested in, and then
               apply all of the updates.
25         for (var i = 0; i < list_of_updates.length; i++) {
26
27             var update = list_of_updates[i];
28
29
30
```

```
31          var operation = update.update_type;
32          var update_path = update.update_path.split(':')[0];
33          var aspect = update.update_path.split(':')[1];
34          var component = update.update_data;
35
36          console.log(operation + " applied to " +
               update.update_path);
37          switch(operation) {
38
39              case "INSERT":
40
41                  try {
42
43                      var components = answer[lab][problem][
                           value][state][update_path][aspect];
44
45                      // If we are inserting into a new module,
                            make sure that we are inserting
46                      // into something that is valid!
47                      if (components === undefined) {
48
49                          answer[lab][problem][value][state][
                               update_path][aspect] = [];
50                          components = [];
51
52                      }
53
54                      components.push(component);
55                      answer[lab][problem][value][state][
                           update_path][aspect] = components;
56
57                  } catch (e){
58
59                  }
60
61                  break;
```

```
62
63                  case "DELETE":
64
65                      try {
66
67                          var components = answer[lab][problem][
                               value][state][update_path][aspect];
68
69                          answer[lab][problem][value][state][
                               update_path][aspect] = refactor_array(
                               component, components);
70
71                      } catch(e) {
72
73                      }
74
75                      break;
76
77                  case "UPDATE":
78
79                      try {
80
81                          var components = answer[lab][problem][
                               value][state][update_path][aspect];
82                          var splice_index = component[
                               component.length - 1];
83
84                          components[splice_index] = component;
85                          answer[lab][problem][value][state][
                               update_path][aspect] = components;
86
87                      } catch(e) {
88
89                      }
90
91                      break;
```

```
92
93                  case "MODULE_EDIT":
94                      try {
95                          if (!answer[lab][problem][value][state][
                               update_path]) {
96                              answer[lab][problem][value][state][
                                   update_path] = {};
97                          }
98
99                      } catch(e) {
100
101                     }
102
103                     break;
104
105                 case "MODULE_DELETE":
106
107                     try {
108                         delete answer[lab][problem][value][state
                               ][update_path];
109                     } catch (e) {
110
111                     }
112
113                     break;
114
115                 case "MODULE_COPY":
116
117                     try {
118                         answer[lab][problem][value][state][
                               update_path] = {};
119                     } catch(e) {
120
121                     }
122
123                     break;
```

```
124
125                 case "RESET":
126
127                     for (candidate in component) {
128
129                         try {
130                             answer[lab][problem][value][state][
                                    candidate]["schematic"] = [];
131                         } catch(e) {
132
133                         };
134
135                     };
136
137                     break;
138
139                 default:
140                     throw new Error("Unhandled update type.");
141             };
142
143         };
144
145         // Update the generation number, will succeed for all
                schematics
146         try {
147             answer[lab][problem][value]["sync_data"].generation =
                    parseInt(generation) + 1;
148         } catch (e) {
149             answer[lab][problem][value]["sync_data"] = {
                    open_jade: 1, generation: parseInt(generation) +
                    1}
150         }
151
152
153
154
```

```
155         // Commit all of the changes
156         req.local.answersX.update({_id: kerberos}, {$set: answer
                }, {upsert: true}, function(err) {
157
158             if (err){
159                 console.log(err)
160             }
161         });
162
163     });
164
165     var response = {
166         timestamp: (new Date).getTime(),
167         generation: parseInt(generation) + 1,
168     };
169
170     res.send(response);
171 });
```

Figure D-7: Route for handing modular updates.

Appendix E

Operational Transform Code

In this appendix, one will find the code listing for the operational transform jade module, split into the following sections:

1. Generalized Operational Transformation Algorithm
2. Inclusion Transform Logic
3. Exclusion Transform Logic
4. Test Server State

```
1  function operational_transform(op, hb) {
2
3    for (i = 0; i < hb.length; i++) {
4
5      if (independent(op, hb[i])) {
6
7        var redundants = [];
8        var cleaned = [];
9
10       for (j = i; j < hb.length; j++) {
11
12         if (causally_preceding(op, hb[j])) {
13           redundants.push(hb[j]);
14         }
15
16       }
17
18       if (redundants.length == 0) {
19         return list_IT(op, hb.slice(i));
20       }
21
22       else {
23
24         var op_temp;
25
26         var causal_op = redundants[0];
27         var index = causal_op.hb_index;
28
29         var excluded_op = list_ET(causal_op, hb.slice(i, index).
               reverse());
30         fixed_ops.push(excluded_op);
31
32         for (k = 1; k < redundants.length; k++) {
33
34           causal_op = redundants[k];
35           index = causal_op.hb_index;
```

```
36
37            op_temp = list_ET(causal_op, hb.slice(i, index).reverse
                 ());
38            excluded_op = list_IT(op_temp, fixed_ops);
39            fixed_ops.push(excluded_op);
40
41          }
42
43          op_temp = list_ET(op, fixed_ops.reverse());
44
45          return list_IT(op_temp, history_buffer.slice(i));
46
47        }
48
49      }
50
51    }
52
53    return op;
54  }
```

Figure E-1: Implementation of the GOT Algorithm.

```
1  function list_IT(op, list) {
2
3    if (list.length == 0) {
4      return op;
5    } else {
6      var corrected_op = inclusion_transform(op, list[0]);
7      return list_IT(corrected_op, list.splice(1));
8    }
9
10 }
11
12 function inclusion_transform(opA, opB) {
13
14   var commandA = opA.operation;
15   var commandB = opB.operation;
16
17   if (commandA == 'INSERT' && commandB == 'INSERT') {
18     return it_ii(opA, opB);
19   }
20
21   else if (commandA == 'INSERT' && commandB == 'UPDATE') {
22     return it_iu(opA, opB);
23   }
24
25   else if (commandA == 'INSERT' && commandB == 'DELETE') {
26     return it_id(opA, opB);
27   }
28
29   else if (commandA == 'UPDATE' && commandB == 'INSERT') {
30     return it_ui(opA, opB);
31   }
32
33   else if (commandA == 'UPDATE' && commandB == 'UPDATE') {
34     return it_uu(opA, opB);
35   }
36
```

```
37    else if (commandA == 'UPDATE' && commandB == 'DELETE') {
38      return it_ud(opA, opB);
39    }
40
41    else if (commandA == 'DELETE' && commandB == 'INSERT') {
42      return it_di(opA, opB);
43    }
44
45    else if (commandA == 'DELETE' && commandB == 'UPDATE') {
46      return it_du(opA, opB);
47    }
48
49    else if (commandA == 'DELETE' && commandB == 'DELETE') {
50      return it_dd(opA, opB);
51    }
52
53    else {
54      throw new Error("Invalid transform type detected.");
55    }
56
57  };
```

Figure E-2: Implementation of the Inclusion Transformation Control Algorithm.

```
1  function it_ii(opA, opB) {
2
3    if (opA.index < opB.index) {
4      return opA;
5    }
6
7    else if (opA.index == opB.index && opA.author > opB.author) {
8      return opA;
9    }
10
11   else {
12
13     var update_data = opA.update_data;
14     update_data[update_data.length - 1] += 1;
15
16     var ot_update = {
17       author: opA.author,
18       update_type: opA.update_type,
19       update_path: opA.update_path,
20       update_data: component_update,
21       index: opA.index + 1,
22     };
23
24     return ot_update;
25   }
26
27 };
28
29 function it_iu(opA, opB) {
30
31   return opA;
32
33 };
34
35
36
```

```
37 function it_id(opA, opB) {
38
39   if (opA.index <= opB.index) {
40     return opA;
41   }
42
43   else {
44
45     var update_data = opA.update_data;
46     update_data[update_data.length - 1] -= 1;
47
48     var ot_update = {
49       author: opA.author,
50       update_type: opA.update_type,
51       update_path: opA.update_path,
52       update_data: component_update,
53       index: opA.index - 1,
54     };
55
56     return ot_update;
57   }
58
59 };
60
61
62
63
64
65
66
67
68
69
70
71
72
```

```
73  function it_ui(opA, opB) {
74
75    if (opA.index < opB.index) {
76      return opA;
77    }
78
79    else {
80
81      var update_data = opA.update_data;
82      update_data[update_data.length - 1] += 1;
83
84      var ot_update = {
85        author: opA.author,
86        update_type: opA.update_type,
87        update_path: opA.update_path,
88        update_data: component_update,
89        index: opA.index + 1,
90      };
91
92      return ot_update;
93    }
94
95  };
96
97  function it_uu(opA, opB) {
98
99    return opA;
100
101 };
102
103 function it_ud(opA, opB) {
104
105   if (opA.index < opB.index) {
106     return opA;
107   }
108
```

```
109    else if (opA.index > opB.index) {
110
111      var update_data = opA.update_data;
112      update_data[update_data.length - 1] -= 1;
113
114      var ot_update = {
115        author: opA.author,
116        update_type: opA.update_type,
117        update_path: opA.update_path,
118        update_data: component_update,
119        index: opA.index - 1,
120      };
121
122      return ot_update;
123    }
124
125    else {
126
127      var ot_update = {
128        author: opA.author,
129        update_type: 'IDENTITY',
130        update_path: undefined,
131        update_data: undefined,
132        index: undefined,
133      };
134
135      return ot_update;
136    }
137
138  };
139
140  function it_di(opA, opB) {
141
142    if (opA.index <= opB.index) {
143      return opA;
144    }
```

```
145
146   else {
147
148     var update_data = opA.update_data;
149     update_data[update_data.length - 1] += 1;
150
151     var ot_update = {
152       author: opA.author,
153       update_type: opA.update_type,
154       update_path: opA.update_path,
155       update_data: component_update,
156       index: opA.index + 1,
157     };
158
159     return ot_update;
160   }
161
162
163 };
164
165 function it_du(opA, opB) {
166
167   return opA;
168
169 };
170
171 function it_dd(opA, opB) {
172
173   if (opA.index < opB.index) {
174     return opA;
175   }
176
177   else if (opA.index > opB.index) {
178
179     var update_data = opA.update_data;
180     update_data[update_data.length - 1] -= 1;
```

```
    var ot_update = {
      author: opA.author,
      update_type: opA.update_type,
      update_path: opA.update_path,
      update_data: component_update,
      index: opA.index - 1,
    };

    return ot_update;

  }

  else {

    var ot_update = {
      author: opA.author,
      update_type: 'IDENTITY',
      update_path: undefined,
      update_data: undefined,
      index: undefined,
    };

    return ot_update;

  };

};
```

Figure E-3: Implementation of the unique sub-cases for the Inclusion Transformation Algorithm.

```
1  function list_ET(op, list) {
2
3    if (list.length == 0) {
4      return op;
5    } else {
6      var corrected_op = exclusion_transform(op, list[0]);
7      return list_ET(corrected_op, list.splice(1));
8    }
9
10 }
11
12
13 function exclusion_transform(opA, opB) {
14
15   var commandA = opA.operation;
16   var commandB = opB.operation;
17
18   if (commandA == 'INSERT' && commandB == 'INSERT') {
19     return et_ii(opA, opB);
20   }
21
22   else if (commandA == 'INSERT' && commandB == 'UPDATE') {
23     return et_iu(opA, opB);
24   }
25
26   else if (commandA == 'INSERT' && commandB == 'DELETE') {
27     return et_id(opA, opB);
28   }
29
30   else if (commandA == 'UPDATE' && commandB == 'INSERT') {
31     return et_ui(opA, opB);
32   }
33
34   else if (commandA == 'UPDATE' && commandB == 'UPDATE') {
35     return et_uu(opA, opB);
36   }
```

```
37
38    else if (commandA == 'UPDATE' && commandB == 'DELETE') {
39      return et_ud(opA, opB);
40    }
41
42    else if (commandA == 'DELETE' && commandB == 'INSERT') {
43      return et_di(opA, opB);
44    }
45
46    else if (commandA == 'DELETE' && commandB == 'UPDATE') {
47      return et_du(opA, opB);
48    }
49
50    else if (commandA == 'DELETE' && commandB == 'DELETE') {
51      return et_dd(opA, opB);
52    }
53
54    else {
55      throw new Error("Invalid transform type detected.");
56    }
57
58  }
```

Figure E-4: Implementation of the Exclusion Transformation Cases.

```
1  function et_ii(opA, opB) {
2
3    if (opA.index <= opB.index) {
4      return opA;
5    }
6
7    else if (opA.index == opB.index && opA.author > opB.author) {
8      return opA;
9    }
10
11   else {
12
13     var update_data = opA.update_data;
14     update_data[update_data.length - 1] += 1;
15
16     var ot_update = {
17       author: opA.author,
18       update_type: opA.update_type,
19       update_path: opA.update_path,
20       update_data: component_update,
21       index: opA.index + 1,
22     };
23
24     return ot_update;
25   }
26
27 };
28
29 function et_iu(opA, opB) {
30
31   return opA;
32
33 };
34
35 function et_id(opA, opB) {
36
```

```
37    if (opA.index <= opB.index) {
38      return opA;
39    }
40
41    else {
42
43      var update_data = opA.update_data;
44      update_data[update_data.length - 1] -= 1;
45
46      var ot_update = {
47        author: opA.author,
48        update_type: opA.update_type,
49        update_path: opA.update_path,
50        update_data: component_update,
51        index: opA.index - 1,
52      };
53
54      return ot_update;
55    }
56
57  };
58
59  function et_ui(opA, opB) {
60
61    if (opA.index < opB.index) {
62      return opA;
63    }
64
65    else {
66
67      var update_data = opA.update_data;
68      update_data[update_data.length - 1] += 1;
69
70      var ot_update = {
71        author: opA.author,
72        update_type: opA.update_type,
```

```
73        update_path: opA.update_path,
74        update_data: component_update,
75        index: opA.index + 1,
76      };
77
78      return ot_update;
79    }
80
81  };
82
83  function et_uu(opA, opB) {
84
85    return opA;
86
87  };
88
89  function et_ud(opA, opB) {
90
91    if (opA.index < opB.index) {
92      return opA;
93    }
94
95    else if (opA.index > opB.index) {
96
97      var update_data = opA.update_data;
98      update_data[update_data.length - 1] -= 1;
99
100     var ot_update = {
101       author: opA.author,
102       update_type: opA.update_type,
103       update_path: opA.update_path,
104       update_data: component_update,
105       index: opA.index - 1,
106     };
107
108     return ot_update;
```

```
109     }
110
111     else {
112
113       var ot_update = {
114         author: opA.author,
115         update_type: 'IDENTITY',
116         update_path: undefined,
117         update_data: undefined,
118         index: undefined,
119       };
120
121       return ot_update;
122     }
123
124   };
125
126   function et_di(opA, opB) {
127
128     if (opA.index <= opB.index) {
129       return opA;
130     }
131
132     else {
133
134       var update_data = opA.update_data;
135       update_data[update_data.length - 1] += 1;
136
137       var ot_update = {
138         author: opA.author,
139         update_type: opA.update_type,
140         update_path: opA.update_path,
141         update_data: component_update,
142         index: opA.index + 1,
143       };
144
```

```
145      return ot_update;
146    }
147
148
149  };
150
151  function et_du(opA, opB) {
152
153    return opA;
154
155  };
156
157  function et_dd(opA, opB) {
158
159    if (opA.index < opB.index) {
160      return opA;
161    }
162
163    else if (opA.index > opB.index) {
164
165      var update_data = opA.update_data;
166      update_data[update_data.length - 1] -= 1;
167
168      var ot_update = {
169        author: opA.author,
170        update_type: opA.update_type,
171        update_path: opA.update_path,
172        update_data: component_update,
173        index: opA.index - 1,
174      };
175
176      return ot_update;
177
178    }
179
180    else {
```

```
181
182       var ot_update = {
183         author: opA.author,
184         update_type: 'IDENTITY',
185         update_path: undefined,
186         update_data: undefined,
187         index: undefined,
188       };
189
190       return ot_update;
191
192     };
193
194   };
```

Figure E-5: Implementation of the unique sub-cases for the Inclusion Transformation Algorithm.

```
1  // Create the express app, pointing to public for the html
2
3  var express = require('express');
4  var app = express();
5  var server = require('http').createServer(app);
6
7  var path = require('path');
8  app.use(express.static(path.join(__dirname, '/public')));
9
10 server.listen(8000);
11
12 // Enable sockets
13 var io = require('socket.io')(server);
14
15 // Create a namespace for circuit editing
16 var schematics = io.of('/schematics');
17
18 // Give authors an ID
19 var author_id = 0;
20
21 schematics.on('connect', function(socket) {
22
23   // Gives the user a unique author number for use in tie
       breakers
24   socket.emit('author_id', {author: author_id});
25   author_id++;
26
27   // Handles changes in the schematic and broadcasts them
28   socket.on('schematic_update', function(data) {
29     console.log(data);
30     socket.broadcast.emit('broadcast_update', data);
31   });
32
33 });
```

Figure E-6: Simple backend for OT-Enabled JaDE.

Appendix F

Generalized Operational Transform Examples

As seen in Chapter 4.3.4, the generalized operational transform has three distinct cases that it needs to handle. In this appendix, there are worked out examples which go through these distinct cases, including timing diagrams, detailed accounts of how the history buffer is maintained, and which fundamental transforms are used in ensuring state coherence. For these examples, I will be considering various workflows that may arise in the construction of a Full Adder unit.

To reiterate, the three cases which the GOT algorithm handles are as follows:

Case 1: In the first case, all updates applied are due to the local user. In fewer words, the default case handles a non-collaborative mode.

Case 2: In the second case, the local user user makes any number of edits, and then comes along any number of edits a remote user made. At this point, the local user never makes anymore edits.

Case 3: In the third case, which is the most realistic one, there is a mixture of local edits followed by remote edits followed by more local edits and so forth.

Case 1 Example

To begin these series of examples, let us consider two collaborators, Red and Blue. Red and Blue are working on the development of a Full Adder, starting from the predefined initial state and working toward a working system. This implies that both Red and Blue start out with an empty History Buffer, although that is soon to change. Red starts off by putting in two XOR gates to calculate the sum bit, and in Figure F-1 we see a timing diagram of how these edits are made to the document state. Since Red is the active author, he applies the updates immediately without worrying about transformation to his own document state and logs the changes in his history buffer. Remember that operational transforms are not applied to self-generated updates when they are applied to the local state. However, in this scenario, Blue is receiving updates, so an operational transformation may be necessary on his end. As blue receives the updates, he scans his history buffer (which was empty) to find that there were no independent operations that had occurred, and as a result, the GOT algorithm makes no transformation, and the operations are in a sense applied at face value and a record is stored in the history buffer.

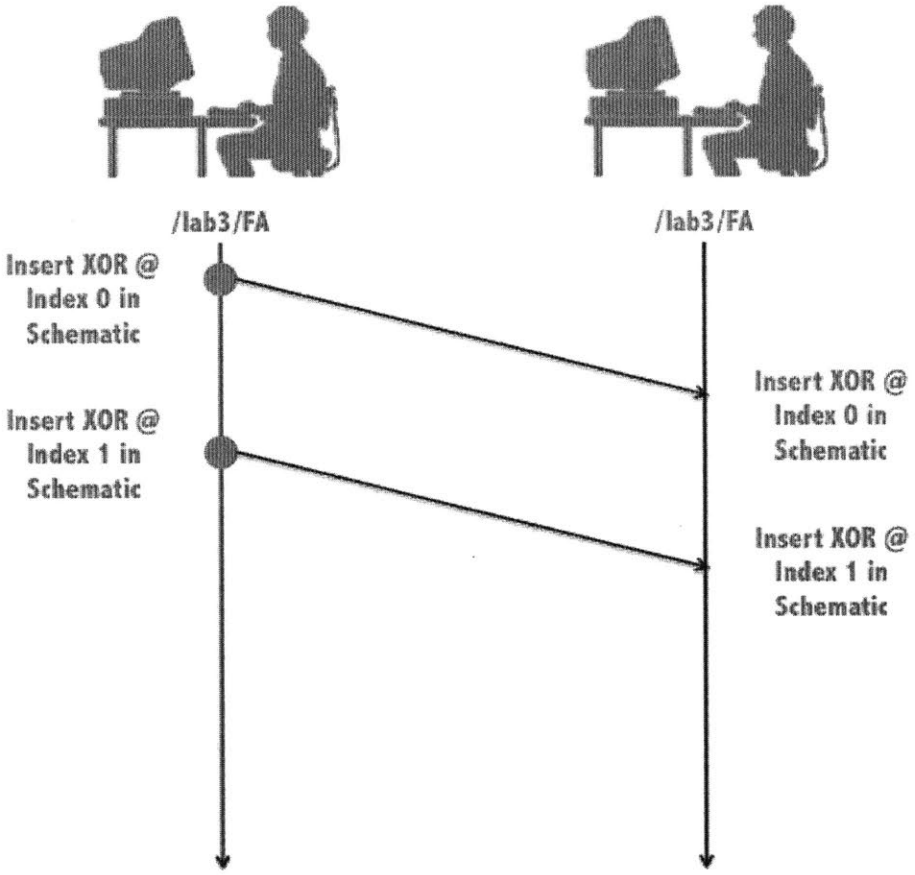


Figure F-1: In this scenario, there are no independent documents updates, and as a result the GOT algorithm applies no transformation.

Case 2 Example

Building on the previous example, lets first take an inventory of the history buffers. For both parties, the history buffer looks like the following:

```
HB = [INS[0], INS[1]]
```

As you can see it contains a total history of everything that has happened. Now, let us imagine what would happen to the history buffer and the document state if we were to now collaborate simultaneously on the same document state. For this, lets assume that blue and red both attempt to insert an and gate, as they wish to construct the logic for calculating the carry out bit. Immediately following their insertions, but before they receive word that the other party has made an insertion, the history buffers now look like the following for Red and Blue respectively:

```
HB = [INS[0], INS[1], INS[2]]
HB = [INS[0], INS[1], INS[2]]
```

In Figure F-2 is a timing diagram illustrating how these updates are applied. What is striking in this scenario, is that there are two attempts to insert at the same location in the array data structure. Now we have to ask ourselves: how do we determine a coherent final state? In these sort of tie-breaker scenarios, we need to have every editor given a priority, with higher priority edits being inserted before the lower priority. In this schema, Red has a higher priority. As Red and Blue receive the update from the other party, they scan their history buffers, ignoring the first two entries since they are causal edits, but upon seeing the last edit (an insertion at index 2), the algorithm recognizes that the operations were independent, and an operational transform takes place. On Red's end, it realizes that the inserts are conflicting, and given that Red has a higher priority, it modifies Blue's edit index to be pushed to index 3. Likewise, Blue will recognize the same thing, but because Blue is of lesser priority, it inserts the update at index 2 in the schematic data structure, pushing everything at index 2 and beyond over one. Therefore, both parties end up with the same schematic data structure![1] Moreover, the intention is preserved and we very nicely avoid having to worry about any other details of the gates through this indexing scheme with operational transforms. Now that we have completed the operational transforms, the resultant updates are saved in the host history buffers as follows:

```
HB = [INS[0], INS[1], INS[2], INS[3]]
HB = [INS[0], INS[1], INS[2], INS[2]]
```

[1]You may notice that the history buffers are different; that is permissible and expected. The document state (in this case, the schematic) is the only thing that needs to converge.

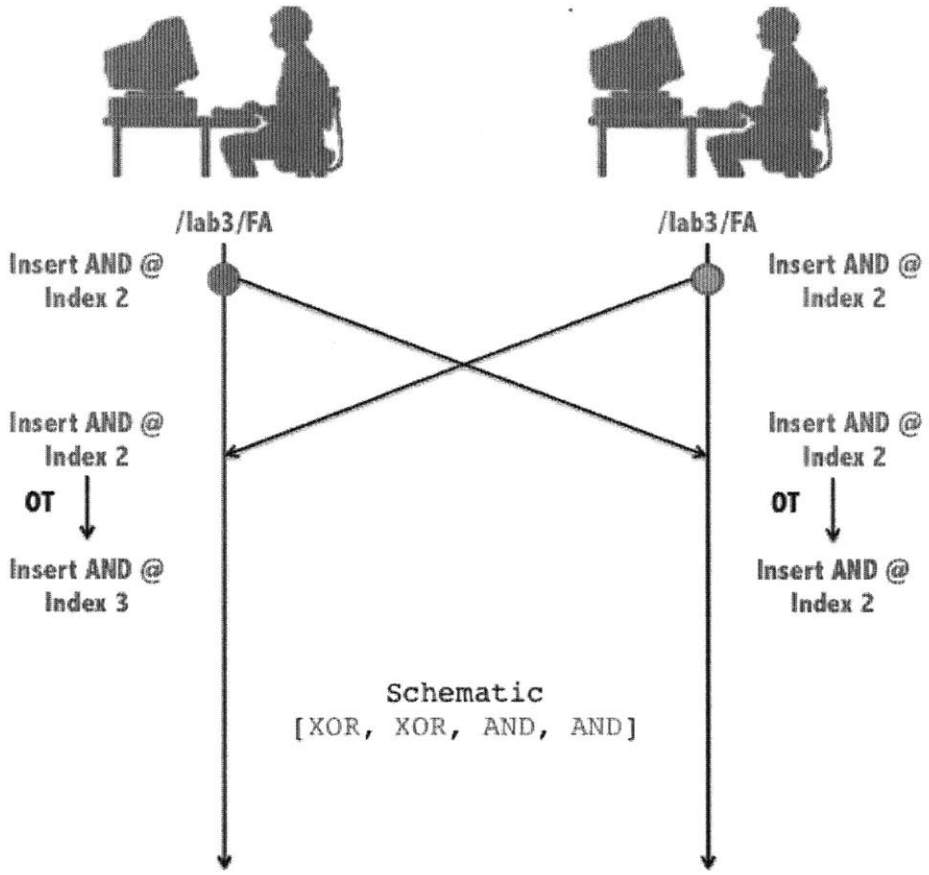


Figure F-2: In this scenario, there are independent documents updates, and as a result the GOT algorithm applies transformation given an arbitrary order.

Case 2 Extensions

In addition to the insert-insert resolution, we also need to consider the examples of when we are dealing with independent operations of different types. In the schema used in this thesis, there are operations of types INSERT, UPDATE, and DELETE. Every permutation of operations requires a different handling mechanism, and they are outlined in Appendix E, Figure E-2. To understand this code, note that the author property is used to derive priority, and the index property is used to denote the index at which an operation is applied. Additionally, here it important to remember that Case 2 deals exclusively with Inclusion Transformations.

Case 3 Example

Although for two-user examples, everything is fine and dandy, issues begin to arise when we generalize to an N-person collaboration. The new issue that arises is a form of over-counting existing updates, which we can visualize in Figure F-3. Let us consider the following timing diagram, with three different users: Red, Blue, and Green.

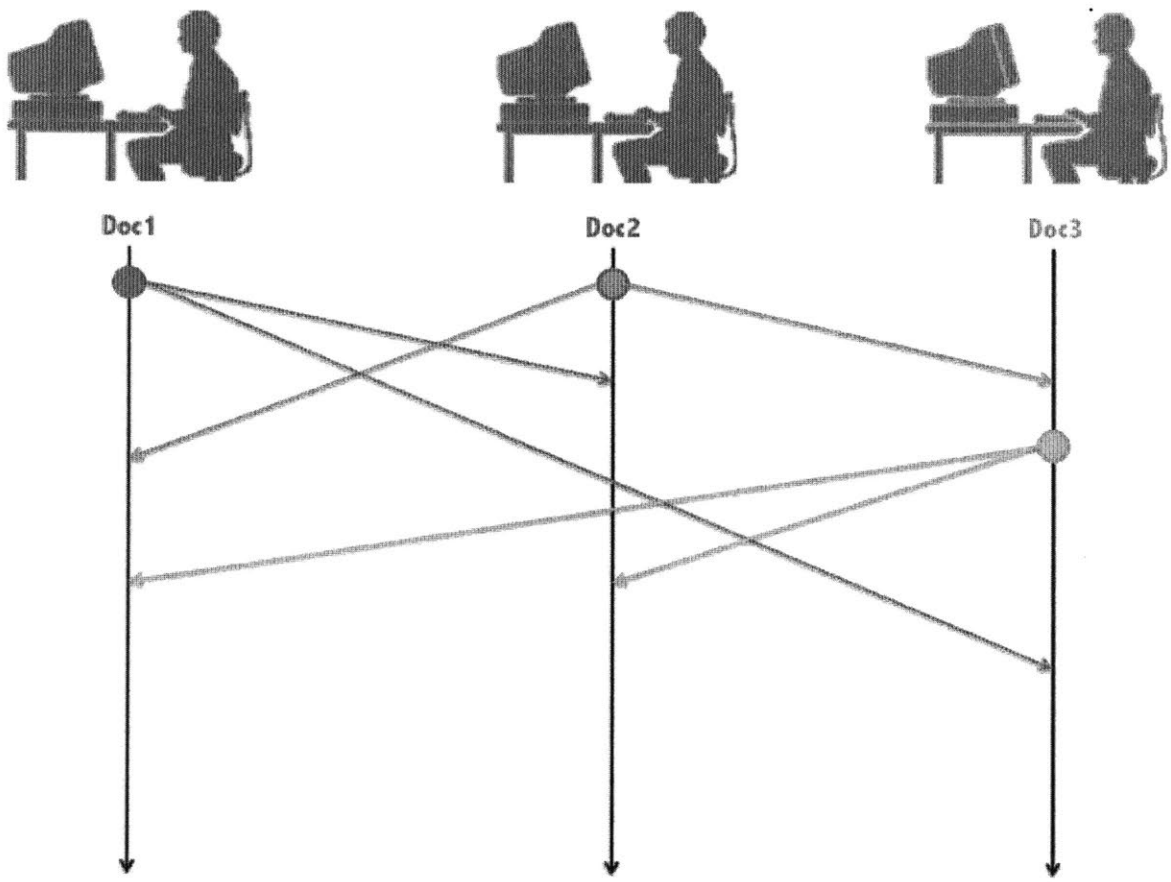


Figure F-3: In this scenario, Red's update arrives at Green late. The update should ignore the effect of green if it wants to assure convergence.

In this scenario, Green would end up with an incorrect state because its own update snuck in when it shouldn't have. Through an exclusion transform, this problem is averted since it allows the system to ignore issues with latency and update the data structure as it should be. On the next two pages, there are examples of what happens without ETs (Figure F-4) and with with ETs (Figure F-5).

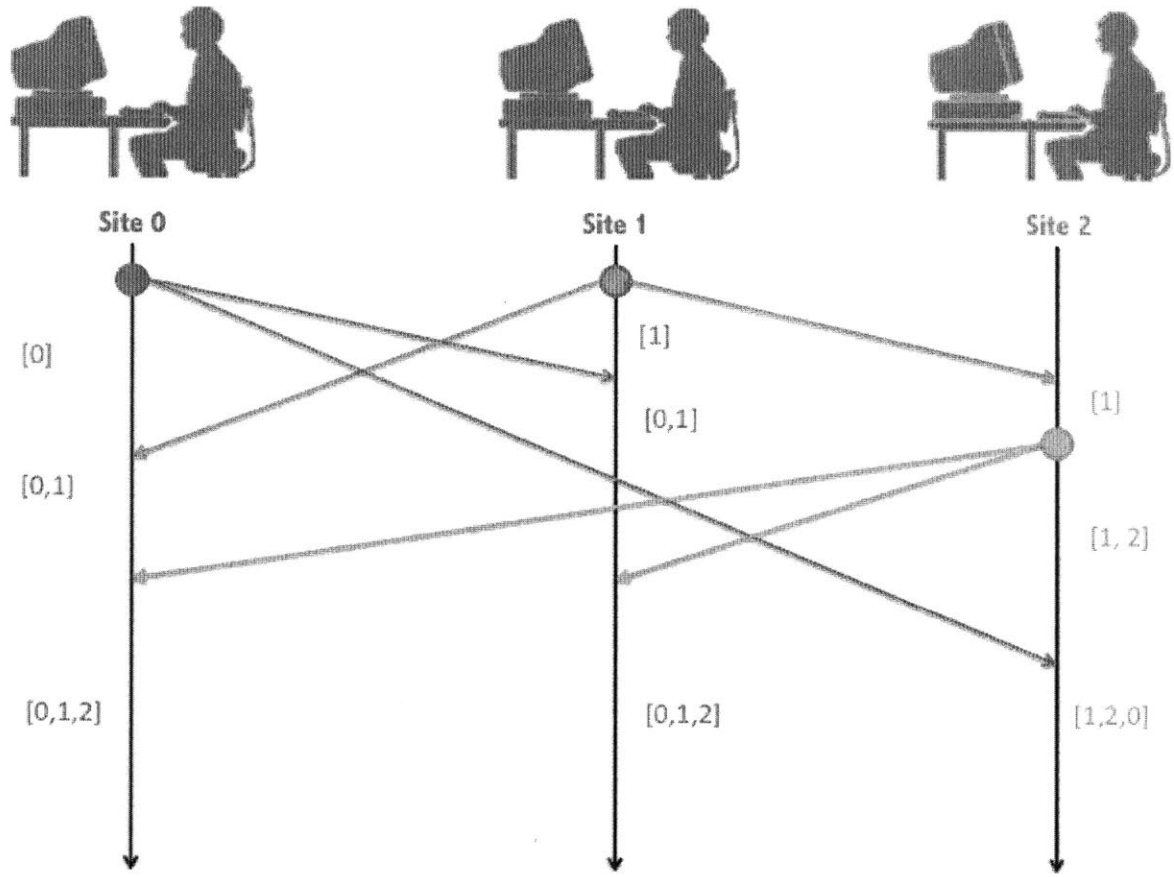


Figure F-4: Example showing how the convergence property is violated without Exclusion Transformations. Notice that Green ends with a dissimilar state because it couldn't dissociate the effect of the Green update from the Red update.

In this example, all users are simply trying to append to the end of the schematic data structure. While this is not a problem for two of the collaborators, Green runs into trouble even though he has the information to figure out how to fix the problem. Let us examine what happened to Green. Well, as it received the Blue Update, there was nothing to transform against, so Blue's element was inserted without question. Then Green made a causal update, which does not require an operational transformation. However, Red's update independent to Blue's and preceding Green's arrives late to Green! In this case, Green should have looked into the metadata associated with Red's update, determined that its own update came after Red's, and when scanning the history buffer, exclude the effect of the Green update from the effect of the Red update. Were this to happen, the Red Update would have an inclusion inclusion

transform applied against Blue's update, yielding [0, 1, 2] which effectively ignores Green's temporally later update to yield a coherent state.

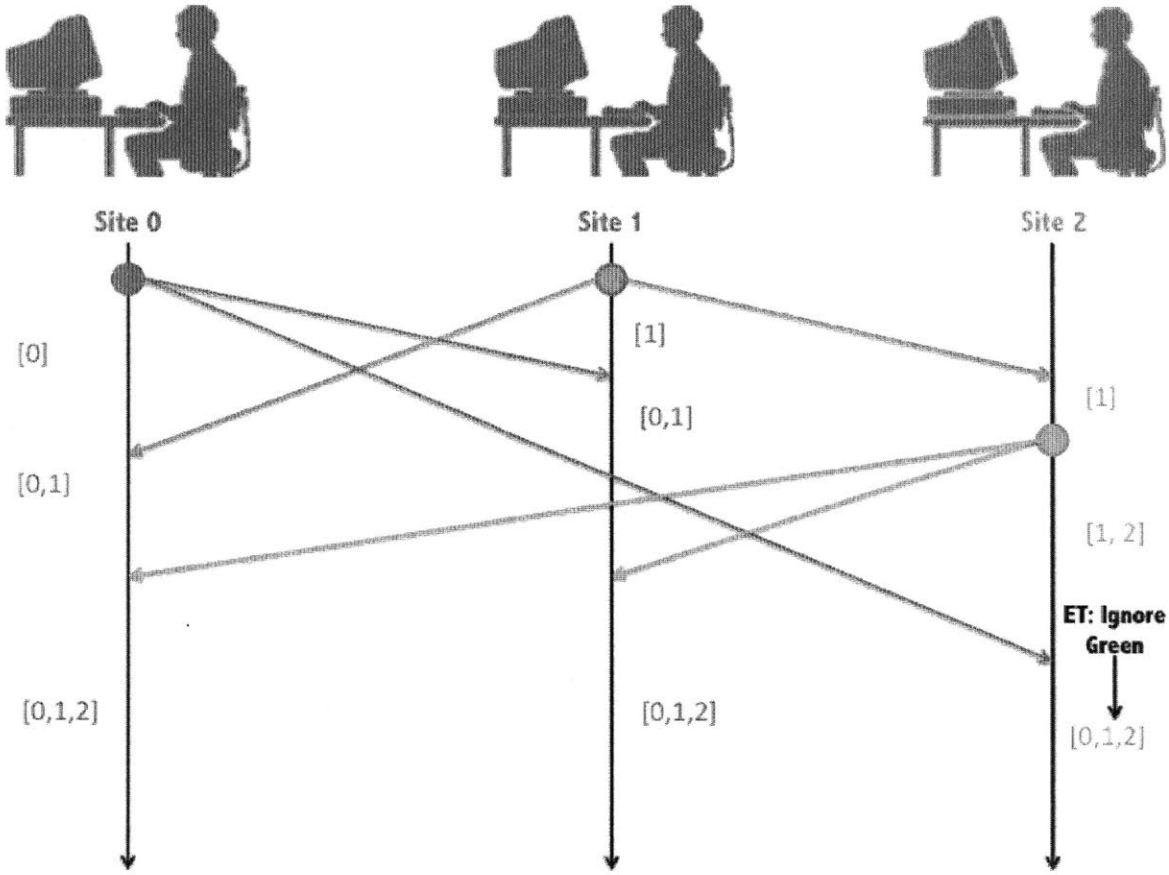


Figure F-5: Example showing how the convergence property is conserved with Exclusion Transformations. Upon getting the Red update, the algorithm recognizes that it is a late arrival through attached metadata, and subsequently ignores the effect of the Green update which should have come later. Through this exclusion transform, all editors can converge to a single coherent state.

Bibliography

[1] http://www.emarketer.com/article/with-mature-us-online-population-small-gains-email-search-usage/1009704, 2016.

[2] Gangadhar Konduri. A collaborative environment for distributed web-based cad, 1997.

[3] JCR Licklider. The computer as a communication device, 1968.

[4] Clarence Sun, ChengzhengEllis. Operational transformation in real-time group editors: Issues, algorithms, and achievements, 1998.