Chair of
Software Engineering

News and Notification: Propagating Relevant Changes to Developers

Software Engineering Laboratory: Open Source Eiffel Studio

By: Christopher Dentel

Supervised by: Christian Estler
Dr. Martin Nordio
Prof. Dr. Bertrand Meyer

Student Number: 11-909-868

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

ABSTRACT

In any software development team staying up to date on the current changes that are occurring in the project is mandatory. There exist many forms today through which that is accomplished: standup meetings, mailing lists, configuration management logs, and many more. However, a lot of information is proliferated naturally as a result of developers working on projects usually being co-located. However when these developers move into a distributed environment, this natural information proliferation cannot occur, and much more attention must be placed on keeping each other up to date on the changes that are occurring. In this report, a notification system that delivers notifications in real-time as well as notifications that were sent while offline was designed and implemented. The system is well designed, highly extendible, server push driven and tailor-maid especially for the Integrated Development Environment and toolset Cloudstudio. Additionally, two widgets were also created for displaying the notifications in the system.

Table of Contents

Cloudstudio 6
Motivation 6
Features 6
Details of Implementation 7
Cloudstudio: News and Notifications 9
Motivation 9
Final product: Developer perspective 9
Final product: Cloudstudio tool developer perspective 11
Implementation 13
Issues 15
Notification arrival race condition 15
Firing events on load 15
Effectiveness/Analysis 16
Future Work 16
Resolving onLoadEvent 16
Simplify implementation process 16
Notification priority and further grouping 17
Notification delivery mechanisms 17
Table of Figures 18
References 19

Cloudstudio

Motivation

For most large projects and teams, using an Integrated Development Environment (IDE) is a necessity of software development. But traditional desktop software IDEs do not seek to accommodate teams who are spread out across multiple countries, time zones, languages, and cultures. These are the challenges faced by those organizations which pursue global software development (GSD). The challenges that GSD poses are not necessarily new, and approaches to alleviate the difficulties that come with GSD have been previously investigated with many different approaches. Some investigations have focused on comparing different project management approaches, including agile vs. structured development [1]. Carmel and Agarwal [2] investigated means to reducing the "distance" between teams (national, organizational, cultural, and temporal distances) and reducing collaboration. Several other investigations have not sought to avoid collaboration but instead focused on how to better facilitate collaboration across time-zones [3] [4] [5]. One such study [6] utilized the Distributed and Outsourced Software Engineering course (DOSE), [7] [8] using some of the technologies presented in this paper.

Existing IDEs rely on configuration management that leads to disparities in information awareness, such as discovering at commit that two major refactorings have occurred simultaneously. While this style of configuration management seeks to isolate developers from the changes that other developers are making, Cloudstudio[a] seeks to share information in real-time. As online document collaboration websites are eliminating the need to email documents of varying revisions back and forth, Cloudstudio seeks to allow developers to work simultaneously on a project, sharing information between themselves as they like, while also maintaining the isolation that traditional configuration management affords.

Features

Cloudstudio [9] is a web based IDE, allowing developers to access their projects at any time from any machine. This move to the web eliminates the need to maintain and update different versions of software on local machines, while also allowing developers to work where they want, when they want. But Cloudstudio is not just a web-app clone of an existing IDE. Cloudstudio seeks specifically to meet the needs of developing in distributed environments through smarter configuration management and tool integration. While still only a web-app, Cloudstudio integrates development tools, collaboration tools, and verification tools. Some such tools are listed below.

Development:

- *Languages* – Cloudstudio supports projects in Eiffel, Java, C#, and JavaScript.
- *Configuration management* – Cloudstudio's configuration management system encourages developers to share early and share often. Developers commit to their own private branches and chose to share their changes when they wish.

[a] Try out Cloudstudio at: http://www.cloudstudio.ethz.ch

- *External Development* – Cloudstudio's configuration management system allows developers who do not wish to use Cloudstudio to still contribute to projects. As Git is the underlying backend for the project, developers can directly connect with the repository and work without being bound to the IDE.
- *Import / Export projects* – Existing projects can be imported into Cloudstudio and existing projects can be retrieved.
- *Monitors* – Monitors provide an early warning system for developers and allow them to keep track of the changes occurring in a project that are important to them [10].

Collaboration:

- *Chat / Skype* – Cloudstudio allows developers to see what other developers are currently working on the same project, and provides access to both chat and Skype from the IDE.
- *Code Reviews* – Code reviews are fully integrated into the IDE, allowing developers to invite their team members to discuss changes without leaving the IDE.
- *Notifications* – All tools have access to a news / notification system, which keeps developers up to date on what is going on in their project. The system is highly customizable, and is discussed in detail in this report.
- *Document Sharing (In progress)* – Teams will be able to collaborate on non-code documents, and see each other's results in real-time.

Verification and Testing:

- Auto Proof – Auto Proof is a static verification tool for Eiffel which allows for proving Eiffel programs in the browser without the need for any additional specifications [11] [12]. Postconditions are tested against possible preconditions to determine if there are cases in which satisfactory preconditions yield unsatisfactory post conditions.
- *Auto Test* – An entirely automated unit-testing suite which infers tests based off contracts [13] [14]. Developers select how long they wish to run the suite for, and Auto Test exercises the classes to test the bounds of the contract.
- *Auto Fix (integration with Cloudstudio in progress)* – While Auto Test tests the bounds of the contracts of a given class and reports failures, Auto Fix will attempt to generate fixes for the errors found [15] [16]. It uses a combination of both static and dynamic analysis to generate fixes, and then regression tests the fixes to determine if they are a suitable candidate to fix the error found.

Details of Implementation

Cloudstudio is developed using Google Web Toolkit and is deployable as an app-engine app. Cloudstudio's editor is an Eiffel program which has been compiled to JavaScript via an Eiffel to JavaScript compiler developed by Alexandru Dima [17]. For back-end data storage, Cloudstudio uses MySQL.

Cloudstudio is being developed at ETH Zürich by the Chair of Software Engineering. Cloudstudio's principal members include Professor Bertrand Meyer, Dr. Martin Nordio, and

Christian Estler. Over 13 masters and bachelors students from several universities have also been involved in implementing Cloudstudio.[b]

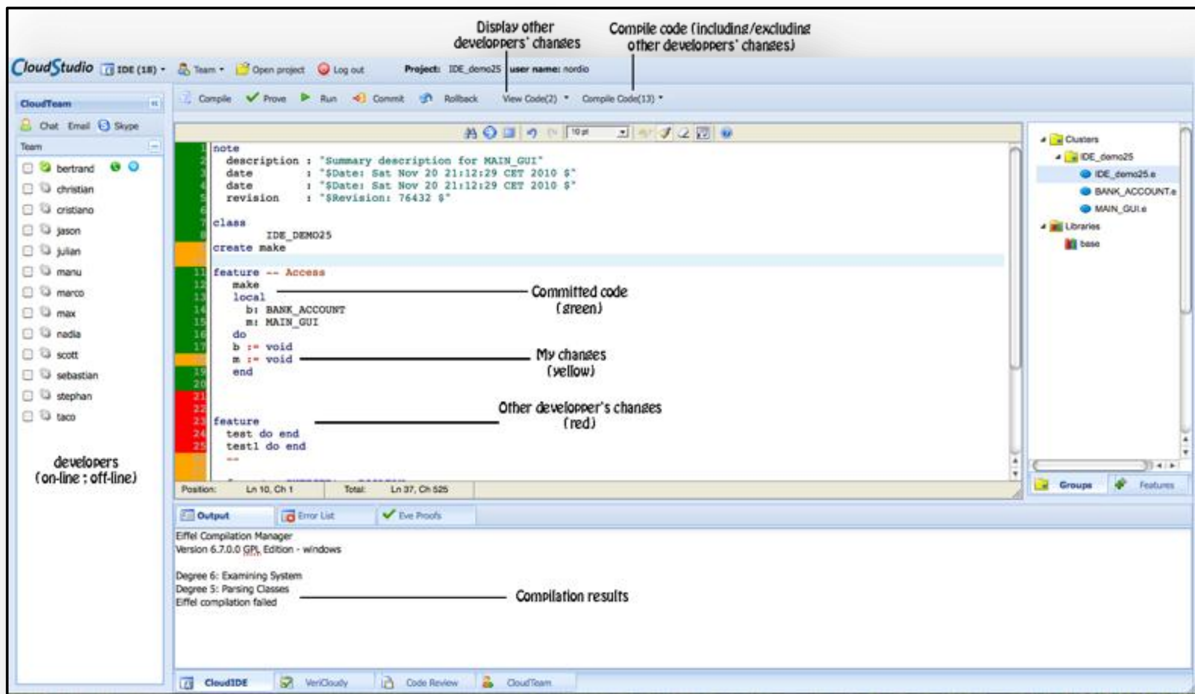

Figure 1: Cloudstudio IDE view.[c]

As is evidenced from the previously mentioned features, Cloudstudio seeks to expand upon the functionalities which are necessary for development while also adding in the features that promote the collaboration necessary to be successful in a distributed development environment. Tools like chat integration, integrated code-reviews, notifications, and monitors bridge the gap that occurs when knowledge cannot naturally circulate through teams that are centrally located. The features in Cloudstudio are tightly coupled yet also flexible, allowing developers to take advantage of the features that benefit them, while not shackling them to the entire suite of tools.

[b] To learn more about the Cloudstudio development effort, visit: http://se.inf.ethz.ch/research/cloudstudio/
[c] Graphic from: http://se.inf.ethz.ch/research/Cloudstudio/

Cloudstudio: News and Notifications

Motivation

In any software development team staying up to date on the current changes that are occurring in the project is mandatory. There exist many forms through which that is accomplished: standup meetings, mailing lists, configuration management logs, and many more. Typically, a lot of information is proliferated naturally as a result of developers working on co-located projects. However when these developers move into a distributed environment, this natural information proliferation cannot occur, and much more attention must be placed on keeping each other up to date on the changes that are occurring.

As most Integrated Development Environments are not designed with web connectivity in mind, creating a part of the IDE that is entirely dedicated to notifying users of recent changes in their projects has not been explored. However, with an IDE that resides fully in the cloud, creating a notification system that collects and distributes notifications from each tool in the IDE can become a reality. In this report, a notification system that delivers notifications in real-time as well as notifications that were sent while offline was designed and implemented.

The final notification system is well designed, highly extendible, serve- push driven and tailor-maid especially for Cloudstudio. Additionally, two widgets were also created for displaying the notifications in the system.

As this system was designed to extended by other developers of Cloudstudio, the final system will be described and presented from both a developer perspective (those who are using Cloudstudio to develop their projects) and from the Cloudstudio tool developer perspective (those who are adding tools to Cloudstudio)

Final product: Developer perspective

The user experience for developers using the notification system in Cloudstudio is very intuitive and much like most modern day notification systems in web applications. An action that Walt performs causes notifications to be delivered to Jesse and Mike. There are two possible cases to account for when delivering these notifications. In the first case the receiving users are online, and in the second they are offline. Both cases have different delivery semantics.

For case one where the receiving users are online, the notifications are delivered almost instantaneously. The developer is informed that they have a new notification through three different areas, each of which has different visibility. The most easily visible notification method is a small info-dialog popup which appears in the lower right hand-corner of the web application (Figure 2). The popup informs the developer that they have a new notification, and the description of the notification is displayed. After a short while, the notification melts away. In the event that the receiving user is not online, they would not experience this popup dialog box upon logging on.

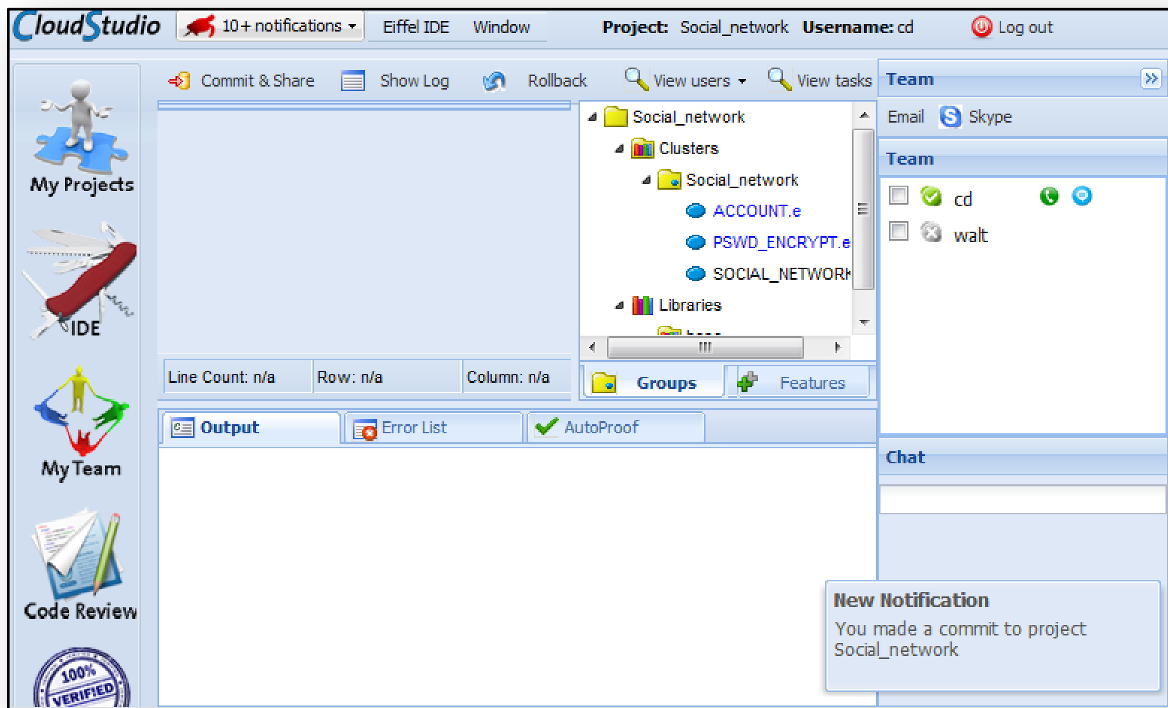

Figure 2 Notification Popup Window

Slightly less visible is the change of the notification jewel in the top-left hand corner of the screen (again, See Figure 2). This jewel displays a blue turtle when there are no active notifications. However, when a notification arrives, the tortoise changes from blue to red, and the text is changed to show how many notifications are available. Upon clicking on the tortoise or the text, a small scrollable list is expanded (See Figure 3) which shows the active notifications. The total amount of notifications to display is capped at 10, and further notifications can be accessed via a last item in the list, which instructs the user that they can click there to see all notifications. When notifications have been accumulated due to a developer being offline, they are delivered at login.

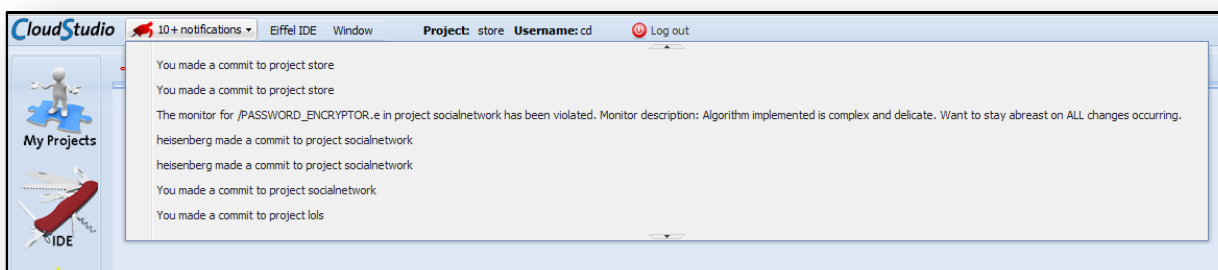

Figure 3 Drop down notification window

Clicking on the previously mentioned button takes users to their news page (Figure 4), where all notifications are displayed. This page allows a developer to see both their active notifications as well as their dismissed notifications by toggling a tab at the top of the display. In both cases, developers are able to highly customize their view, specifying what columns to sort by, what columns they would like to hide, as well as what they would like to group on. In Figure 4, the notifications are grouped by project, but notifications could also be grouped by type (or any other column, though it may not be meaningful).

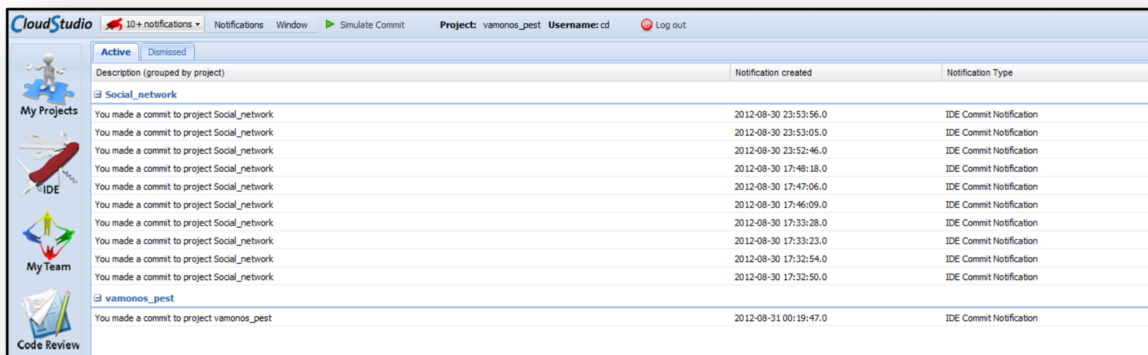

Figure 4 News view, showing active notifications grouped by project

When a developer right clicks on a notification presented in the active tab, they have the option of dismissing the notification. Notifications are not necessarily dismissed when a developer clicks on it from the drop-down box (though the Cloudstudio tool developers can specify this behavior if they wish). Once a notification has been dismissed, it is moved from the active tab to the dismissed tab, and the tortoise jewel is updated to reflect this. Should a notification be received while in the news view, it is immediately added to the view.

These three different presentation methods help keep developers informed at all times. The popup window immediately grabs their attention, the jewel provides a constant reminder, and the news window allows developers to see what is happening across all of their projects in one place. With having to manually dismiss notifications, the news page can almost be used like a to-do list, with each notification being a reminder of something that needs to be done. For instance, the monitor tool previously mentioned uses this notification system to inform users when monitors have been violated. A developer can keep that notification active as long as they still need to resolve the violation. Once the violation has been resolved, the developer (or maybe in a future version the monitor system) can dismiss that notification.

Final product: Cloudstudio tool developer perspective

From a Cloudstudio tool developer's perspective, extending the notification system to support notification types for their tools is simple. While this section will deal with the implementation of the notification system, it will only explain those parts which are necessary to understand how Cloudstudio tool developers would interact with this code.

To add a new notification type to the notification system, developers must complete three steps:

- Define a new model for their notification. This will be what is delivered to the client side code.
- Create a new Database Table which implements a single method: *fetch*. *fetch* returns an instance of the model defined in the first bullet.
- Add methods to create notifications to the notification client-service RPC interface and create a new notification type for this class.

Developers extend the class *NotificationModel* (defined later), and they supply methods for all abstract methods. This is the model which will be delivered to the client when a notification is to be sent. There are some properties required by *NotificationModel* but additional methods and fields can be added to this subclass so long as they meet the serialization requirements of Google Web-Toolkit and the server-push system used (which will also be described in Implementation). This notification has several behaviors that enable the expressiveness of the system. Developers must implement a *onSelected* method, which will be executed when a user either clicks on the notification from the notification drop-down menu or when they click on the notification in the news system. In addition, the developers can also specify that when the notification is loaded that they would like to register an event on the event handler. This feature is still being explored, but would allow for event driven programming across several different front-ends of the web app. There are currently some issues with this that are described in the Issues section and in Further Work.

Developers then create a *TypeOfNotificationTable* which stores all necessary information to build their subclass of *NotificationModel*. The only requirement on this table is that it supports a fetch method which satisfies the interface *TypeOfNotificationTable*, which returns the model that matches a particular id. It is useful for this table to return the generated id of the row when a new notification is added to the table.

To provide access to their new *TypeOfNotificationTable*, developers add a new notification type to the *NotificationServiceImpl* class. They also add any creation methods to the *NotificationServiceImpl* class. This method must both create a new notification in the new table that the developer has made and also call *registerNew* in the *NotificationServiceImpl* class (providing their notification type and the id of the newly created notification). This stores a record in a master notification table that there is a notification that exists in this new developer defined table and it can be looked up using id. To provide access to lookup this notification, the developer adds a function call to *NotificationSericeImpl*s load method, which calls the *fetch* method of their table (Figure 5).

With this, highly expressive and highly customized notifications can be transmitted to the client code. Developers do not need to worry about serialization or delivery mechanisms, but can define the behavior of their notification entirely within their model. In addition to automating the delivery of notifications, the notification system also provides additional features. The notification service developed contains the ability to destroy notifications on the client side which already have been transmitted. This is very useful to de-duplicate notifications that have since been updated. For instance, a notification is sent when the developer Mike makes a commit to the project "Vamonos_Pests." Developer Jesse then makes a commit to that same project. A second notification could be sent informing all developers about this change. But

then Walt, who is also working on "Vamonos_Pests" with Jesse and Mike, would have two notifications which are mostly redundant. If the configuration management system chooses to, they can avoid this by retracting the original notification, and then pushing a single new notification stating that both Mike and Jesse have made commits to "Vamonos_Pests".

The mechanism for dismissing notifications is inherited from *NotificationModel*, and developers can specify in their subclasses that they wish to dismiss a notification when it is first interacted with. Dismissal requires no changes in the developer defined *TypeOfNotificationTable*.

In the event that a developer does not know when they need to add a notification (or perform an operation to check to see if they should add notifications) the developers can use *NotificationArbiter* which is a collection of event listeners that dispatch RPC calls to *NotificationServiceImpl*. For example, the file monitor system listens for *IdeCommitEvent*s which are fired whenever the configuration management is modified. The handler in *NotificationArbiter* handles this event and sends an RPC to *NotificationServiceImpl* requesting that the file monitor system inspect itself for violations and push notifications for all violations found.

Implementation

Some of the implementation is described in the previous section, but this section will expand on this and discuss the implementation that the other Cloudstudio developers do not interact with.

The abstract *NotificationModel* class declares as abstract all methods that the front-end implementation needs in order to display and handle the model correctly.

NotificationServiceImpl provides many methods that the front end code will access. Namely, it provides a method to request all active notifications for a specific user. While the notification system is server-push, all pre-existing notifications must be fetched when the user logs in. This class also offers register, dismiss, and notification recall methods.

While the other Cloudstudio developers are responsible for providing the table capable of creating their models when asked to do so, their tables do not know **when** to create the models and **how** to deliver notifications. This is where the master table described earlier (*NotificationTable)* is used. This table is again rather simple, having only 6 columns (excluding id):

- Project id: What project this notification pertains to?
- User id: What user this notification is for?
- Timestamp: When was this notification last modified?
- Type: What type of notification is this?
- Notification Id: What is the id of the notification in its own table?
- Active: Is this notification active or dismissed?

When the client-side requests all notifications for a particular lookup, each relevant row in the datastore is converted into a *NotificationLookupModel*, which is a simple java object encapsulating a single table row. These *NotificationLookupModel*s are then converted to NotificationModels through the *load* method (see Figure 5).

```
/**
 * String representations of notifications for later look up.
 */
private static final String MONITOR_VIOLATED_NOTIFICATION = "1";
private static final String IDE_COMMIT_NOTIFICATION = "2";
private static final String REVIEW_NOTIFICATION = "3";

public NotificationModel load(NotificationLookupModel nlm) {
        TypeOfNotificationTable table;
        if (nlm.getType().equals(MONITOR_VIOLATED_NOTIFICATION)) {
                table = new MonitorNotificationTable();
        } else if (nlm.getType().equals(IDE_COMMIT_NOTIFICATION)) {
                table = new IdeNotificationTable();
        } else if (nlm.getType().equals(REVIEW_NOTIFICATION)) {
                table = new ReviewNotificationTable();
        } else {
                throw new IllegalArgumentException("The lookup model provided does not have a
corresponding definition assigned in NotificationServiceImpl");
        }
        return table.fetch(nlm.getNotificationId(), nlm.getId());
}
```

Figure 5 Load method which converts NotificationLookupModels into NotificationModels

These NotificationModels are then sent as a response to the RPC call. It is important to note that RPC is the delivery mechanism only for requests for all notifications, which should only occur once per session.

However, there also exists the case where a notification should be sent immediately after it is created. To accomplish this, the *NotificationServiceImpl* keeps track of all developers currently logged in. When a developer logs into Cloudstudio, they *subscribe* to the Notification Service, which begins a new server-push connection for that user. Their user id is then mapped to this server-push connection, as this is the pipe through which all notifications will be delivered. When a notification is registered, the *NotificationServiceImpl* looks at this mapping to determine if that user is logged on. If so, the *NotificationModel* to deliver is wrapped in a wrapper and then sent through the server push connection.

This process is also very similar for when a developer requests that a previously sent model be invalidated. The notification is dismissed on the server side, and a *NotificationAntiModel* is created that contains the *NotificationModel* to be destroyed. This is then pushed down the server-push connection if the user is logged in. When the client side notification code receives this anti-model, it knows to destroy any notification matching the wrapped one.

All active notifications in the system are kept on the client side as a part of the client state, which uses a specially created *NotificationCallbackBus* to manage interactions across the various widgets. Different widgets can subscribe to the *NotificationCallbackBus* providing an implementation of *NotificationCallback* which behaves much like an *AsyncCallback*, but has methods like *onAdd(model)*, *onRemove(model)*, and *onDismiss(model)*. *NotificationCallback* is abstract and provides overridable implementations of each of the three previously mentioned methods but for lists of models. The notification methods described earlier uses this to stay in sync with the notifications as they exist in the client state; however notifications are not updated or removed through a callback like this. To add, remove, or dismiss a notification, methods in the client must be explicitly invoked.

The system used for server-push is GWT-Comet, a comet implementation specifically for Google Web-Toolkit developed by Richard Zschech[d]. It was very suitable for this task.

Issues

Notification arrival race condition

Perhaps the most substantial issue that was encountered in developing the notification system was a race condition which occurs due to the asynchronous nature of loading all active notifications out of the datastore. This load may still be occurring when

- New notifications are received via server-push,
- New anti-notifications are received via server-push,
- Something on the client side requests to remove a notification,
- Something on the client side requests to dismiss a notification,
- Some component is initialized and requests all notification currently part of the client state.

While many of these issues would occur infrequently, this race condition could lead to an inconsistent datastore or operations being handled incorrectly. To combat this, several of the methods involving Notifications were synchronized in *ClientState*, and requests were delayed until the client-side copy of the notifications was fully loaded. The following bullets describe how the front-end system avoids the race condition. (Note, in the following it is assumed that the set of active notifications have been requested, but not yet received.)

- ❖ A request to add a notification is received → notification is added to a queue, where it will then be added once the backlog of notifications has arrived.
- ❖ An anti-notification arrives → notification is added to a queue, where it will be removed once the backlog of notifications has arrived.
- ❖ The client side requests that a notification is removed → notification is added to a queue, where it will be removed once the backlog of notifications has arrived.
- ❖ The client side requests that a notification is dismissed → RPC call requests that the server dismiss that notification. Notification is added to a queue, where it will be removed from the client side once the backlog of notifications has arrived.
- ❖ A component requests the current notifications out of the client state → a *NotificationRequest* is made for the request, and the asynchronous callback provided is given to the *NotificationRequest*. A *NotificationRequest* encapsulates the behavior to perform once all notifications are loaded. Once all notifications are loaded, the callbacks are executed.

The system sufficiently handles the race condition, preventing the front-end from trying to fulfill requests before it is ready.

Firing events on load

One of the features that make *NotificationModel*s so powerful is their ability to initiate action both when they are received as well as when they are selected. However, even when the

[d] Source and documentation available at http://code.google.com/p/gwt-comet/

notifications have been fetched from the datastore (as opposed to arriving as part of a server-push *NotificationWrapper),* this on load event is still executed. This leads to a massive surge of events being fired as all active notifications are brought into the project. Ideally, onLoadEvents should only be executed if the notification has arrived via server-push, and not if it has been fetched the datastore. This is proposed as further work.

Effectiveness/Analysis

For developers using Cloudstudio, the end result is highly effective and very intuitive. Notifications are presented in an easy to understand way, and the different notification mechanisms direct the developer's attention appropriately to their notifications. The widget which displays the Cloudstudio tortoise jewel can be used ubiquitously throughout the app, and provides a visually consistent way for developers to appraise their notifications.

The news view is also highly effective, allowing developers to see all active notifications for all projects in once place. The grouping and sorting features are very powerful, and allow a developer to use their notifications like a task queue.

The notification system also proved to be effective for Cloudstudio tool developers. While this report includes the development of the notification system, another report developed in parallel (the development of file monitors) [18] is a client to this notification system. Having this use case sufficiently guided the development of the notification system, and the end result was very sufficient for the needs of file monitors. As was mentioned earlier, trying to reduce sending multiple nearly identical notifications led to the development of the anti-notifications, and file-monitors were the original source of these identical notifications.

However, as both systems were developed by the same author, it is difficult to say if the notification system can be effectively extended by other developers. As a further refinement, a student from Politecnico di Milano completed a semester project where he implemented code reviews into Cloudstudio. His experiences and an analysis are presented in a report [19] containing several refinements made to the notification and monitoring system.

Future Work

Future work with the monitor system can be done in several different areas, and projects are listed in decreasing order, beginning with the highest impact for estimated effort.

Resolving onLoadEvent

One barrier that kept some developers from implementing notifications of their own was the previously mentioned issue of stored notifications firing their *onLoadEvents* when they are added to the client state after being returned via RPC. An *onLoadEvent* should probably only be executed when the notification arrives via server-push, and revising the system to behave in this manner should be very simple.

Simplify implementation process

While the report author developed notifications for the notification system presented, it would be beneficial to analyze the difficulties that other Cloudstudio tool developers encounter when implementing notifications of their own. Many of the other existing tools could greatly benefit from sharing information via notifications, so there exist plenty of cases where this could

be done. Should there prove to be common points of failure, these points should either be redesigned or better documented. In addition, the notification system can be analyzed without a usability review to determine if it can be simplified or better structured.

Notification priority and further grouping

As the diversity of notifications shared in Cloudstudio grows, the news view will become more and more powerful. With this view, it would be beneficial to give developers greater control over what they see. Giving developers the ability to create their own groups of notifications or to give some notifications a higher priority over others would further enable developers to use the news tool as a one stop landing page showing them both what is new in Cloudstudio, and what it is that needs to be done.

Notification delivery mechanisms

While the notification system developed here is designed for notifying developers who are already logged into Cloudstudio, it has no way of notifying developers when they are not logged in. Adding optional email notifications would be beneficial for informing developers that there is something that they should log in and take a look at. This system could be customizable, with developers specifying what types of notifications they would like to receive email notifications for. Notifications could also be delivered as a digest form. This digest form could also prove to be beneficial for members who are not developers on the team, but who are interested in staying abreast on changes occurring in the project.

Table of Figures

Figure 1: Cloudstudio IDE view. 8
Figure 2 Notification Popup Window........ 10
Figure 3 Drop down notification window........ 10
Figure 4 News view, showing active notifications grouped by project 11
Figure 5 Load method which converts NotificationLookupModels into NotificationModels..... 14

References

[1] H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer and J. Schneider, "Agile vs. Structured Distributed Software Development: A Case Study," in *7th International Conference on Global Software Engineering*, IEEE, 2012.

[2] E. Carmel and R. Agarwal, "Tactical Approaches for Alleviating Distance in Global Software Development," *IEEE Softw.,* vol. 18, no. 2, pp. 22-29, March 2001.

[3] E. Carmel, Global software teams: collaborating across borders and time zones, Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[4] M. Nordio, R. Mitin, B. Meyer, C. Ghezzi, E. Di Nitto and G. Tamburrelli, "The Role of Contracts in Distributed Development," in *Proceedings of Software Engineering Approaches for Offshore and Outsourced Development*, 2009.

[5] J. A. Espinosa, K. Nan and E. Carmel, "Do Gradations of Time Zone Separation Make a Difference in Performance? A First Laboratory Study," in *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2007)*, IEEE, 2007, pp. 12-22.

[6] M. Nordio, H.-C. Estler, B. Meyer, Ghezzi, C. Ghezzi and E. Di Nitto, "How do Distribution and Time Zones affect Software Development? A Case Study on Communication," in *Proceedings of the IEEE International Conference on Global Software Engineering (ICGSE 2011)*, IEEE, 2011.

[7] M. Nordio, R. Mitin and B. Meyer, "Advanced Hands-on Training for Distributed and Outsourced Software Engineering," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[8] M. Nordio, C. Ghezzi, B. Meyer, E. Di Nitto, G. Tamburrelli, J. Tschannen, N. Aguirre and V. Kulkarni, "Teaching Software Engineering using Globally Distributed Projects: the DOSE course," in *Collaborative Teaching of Globally Distributed Software Development - Community Building Workshop (CTGDSD)*, ACM, 2011.

[9] M. Nordio, H.-C. Estler, C. A. Furia and B. Meyer, "Collaborative Software Development on the Web," 2011.

[10] C. Dentel, "Monitors: Keeping Informed on Code Changes," Independent Research, ETH Zürich, 2012.

[11] M. Nordio, C. Calcagno, B. Meyer, P. Müller and J. Tschannen, "Reasoning About Function Objects," in *TOOLS-Europe*, J. Vitek, Ed., Springer-Verlag, 2010.

[12] J. Tschannen, C. A. Furia, M. Nordio and B. Meyer, "Verifying Eiffel Programs With Boogie," in *First International Workshop on Intermediate Verification Languages (BOOGIE 2011)*,

2011.

[13] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio and B. Meyer, "Stateful Testing: Finding More Errors in Code and Contracts," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2011.

[14] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei and E. Stapf, "Programs That Test Themselves," *IEEE Computer,* vol. 42, no. 9, pp. 56-55, 2009.

[15] Y. Pei, Y. Wei, C. A. Furia, M. Nordio and B. Meyer, "Code-Based Automated Program Fixing," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2011.

[16] J. Tschannen, C. A. Furia, M. Nordio and B. Meyer, "Usable Verification of Object-Oriented Programs by Combining Static and Dynamic Techniques," in *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*, 2011.

[17] A. Dima, "Developing JavaScript Applications in Eiffel," Masters Thesis, ETH Zürich, 2011.

[18] C. Dentel, "Monitors: Keeping Informed on Code Changes," Independent Research, ETH Zürich, 2012.

[19] C. Dentel, "Refinements and Git Integration with Notifications and Monitoring," Software Engineering Laboratory: Open Source Eiffel Studio, ETH Zürich, 2012.