An Architecture for Supporting Opportunistic Collaboration

Dayton Ellwanger

April 28, 2017

Abstract

This paper describes an architecture to support opportunistic collaboration and an implementation of it designed for collaboration between students and instructors in computer programming courses. The implementation is scalable and interoperates with existing collaboration tools where possible, and can be easily extended to support other collaboration scenarios. This paper extends it to support online programming environments and an IoT system to demonstrate that the number of reusable components of the original system is proportional to the similarity of the new system.

I. Introduction

This work builds on the trend of looser notions of collaboration and looser work coupling in collaborative systems. Some of the first collaborative systems were designed as *what you see is what I see* (WYSIWIS) systems - all collaborators saw the same screen. This has the advantage that collaborators are always aware of each others' state, but it is inflexible and provides no benefits that go *"beyond being there"* [1] - collaboration could be at least as effectively reproduced if two programmers were collocated and working side-by-side.

This motivated the next generation of collaboration, *mixed-focus collaboration*, which has collaborators switching between individual work and pair or group work [2] [3]. It assumes that users are working on the same artifact, but allows for work on different aspects of it. It requires *awareness mechanisms* since when users are working on different aspects of the shared artifact they need to be caught up on what others have done prior to collaborating.

A specific type of mixed-focus collaboration is *opportunistic* mixed-focus collaboration, which has the switch from individual work to group work triggered by an event (such as two users simultaneously editing interdependent sections of code) rather than being planned.

Still more flexible is *mixed-activity collaboration*, which allows users to work on different artifacts [4].

Many tools exist that support WYSIWIS, mixed-focus collaboration and opportunistic mixed-focus collaboration. This paper describes an architecture to support opportunistic mixed-activity collaboration and an implementaiton of it designed to aid in the instruction of computer programming courses that interoperates with some of these. This paper further generalizes collaboration and considers scenarios outside of the context of programming and outside of human-to-human interaction that the architecture supports.

II. Driving Problem

The motivating problem for the architecture and implementation presented is that of instructors providing help to students in programming courses where the students work in the IDE Eclipse. The work flow corresponding to this problem is shown in figure 1. The students interact with the programming environment (Eclipse) through their programming activity. Sensors collect information both about the programming environment (e.g. the con-

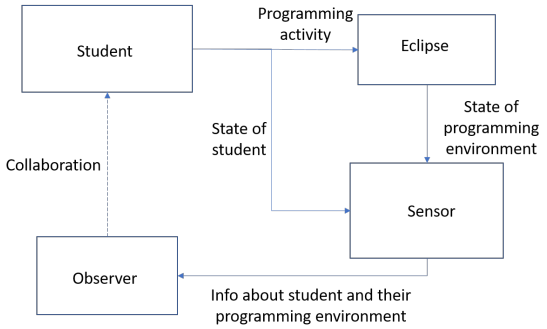

Figure 1: *Driving Problem Work Flow*

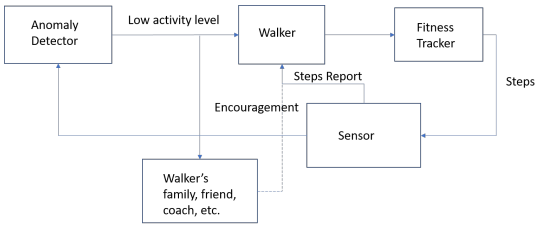

Figure 2: *Fitness Tracker Work Flow*

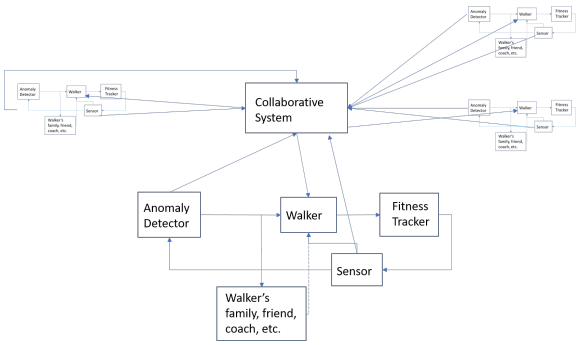

Figure 3: *Networked Loops in the Fitness Tracking Example*

tents of the code editor) and the students (e.g. how much difficulty they are experiencing). The sensors relay this information to the instructors. Based on the information they receive, instructors can choose to collaborate with students.

II.i. Related Problems

The work flow in the driving problem resembles many others. Most obviously, the choice of programming environment does not affect the work flow. Any programming environment could be substituted for Eclipse in figure 1 and the diagram would remain the same.

A less obvious mutation is shown in figure 2. Here the students have been replaced by walkers (i.e. people who walk, and are ostensibly aiming to walk more). The programming environment is replaced with a fitness tracker, which the walkers interact with through their activity. The sensor records how many steps have been taken and reports this back to the walkers. In addition this information is relayed to an anomaly detector that alerts both the walkers and a group of observers when the walkers' activity level is low. This can be viewed as an opportunistic collaboration trigger, where the initiated "work coupling" is the observers offering encouragement to the walkers to return them to the "correct" state - one with higher levels of activity.

This work flow exhibits features that are lacking in the work flow given for the driving problem (figure 1) that could be useful if translated.

Note that the walkers are included in the feedback loop; they are notified of their number of steps and detected anomalies. Perhaps if students in programming courses received information about their programming activity (e.g. how much time they spend programming) they would be encouraged to lead a healthier programming life.

Another important feature of the fitness tracker work flow is that the observers can be other walkers. This introduces the notion of *networked loops*: the walkers can exist in their own loops while serving as observers to other walkers' loops. This is shown in figure 3. Two walkers being aware of each others' activity is a type of work coupling and can serve to drive both towards the correct state. This idea has an obvious extension to the driving problem. Making students in a class aware of each others' programming activity can drive them towards the "correct" state - one where they complete their work on time and correctly.

A final example of a related work flow is shown in figure 4. This example is unique because the actor is non-human; it is an HVAC

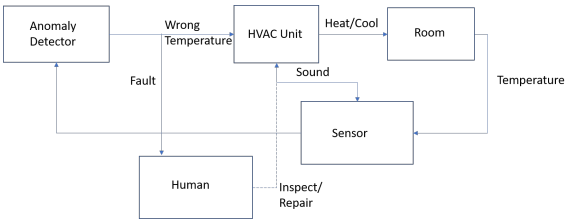

Figure 4: *HVAC Collaboration Loop*

unit. Despite this, the collaboration work flow is roughly the same, which demonstrates the generality of this type of interaction. The HVAC affects a subject, the room or building, by its action of heating or cooling. Sensors detect both the state of the room - its temperature - and the state of the HVAC Unit - in this example through monitoring the noise it makes. Feedback goes both back to the unit, by means of telling it whether it has made the room too hot or too cold, and to a human observer. The trigger for opportunistic collaboration is a notification that the unit is faulty (this would be triggered if the unit were making a lot of noise, for example). Upon receiving this notice, a human observer joins the loop to inspect the unit and repair it if it is broken.

III. Architecture and Implementation Goals

These examples indicate several properties the architecture and implementation should have.

They should be reusable in other scenarios to the extent that the two are similar, which will be referred to as *proportional reuse*. The architecture describes a collaborative system, and it should be modular in the sense that other collaborative systems can use the relevant components of it without much or any modification. Similarly, an opportunistic collaboration application should be able to use the collaboration tools from the implementation as well as the opportunistic collaborative tools. Most specifically, a collaboration tool designed to be used for programming should be able to reuse almost all components of the implementation. In general, the more similar the two systems, the larger the number of shared components between them. Additionally, the implementation should be easily extendable so that it can be modified to meet the needs of other similar problems (an example would be collaborative programming in a work environment rather than an educational one). So alternatively, the more similar the two applications, the fewer the number of modifications needed to morph the implementation to the needs of the new scenario.

In the same way that parts of the implementation should be reusable in scenarios other than the driving problem, it should also reuse components from other systems. For example, it is an instance of a collaborative system, and many tools exist that facilitate collaboration (e.g. email, screen sharing and instant messaging clients). It should *interoperate* with these as much as possible. Interoperability allows for separation of concerns. For example, plenty of software exists specifically designed to tackle the challenging problem of supporting communication between a large number of clients. Rather than solving this and other difficult problems, it is better to leverage existing technology.

The final goal is *scalability*. The implementation should scale to allow anyone on the internet to collaborate with anyone else on the internet. This is a powerful way to satisfy the "beyond being there" requirement. Internet communities such as Stack Overflow dedicated to helping their members solve programming problems have been enormously successful. In the same way, anyone who wants to act as an observer of students' loops should be able to (provided the student allows this - privacy and security will be discussed later).

This goal is also impactful for related scenarios. Imagine an environmental agency sets up IoT devices in streams around the country to gather data on water quality. Eventually these devices will require a change of batteries. Having agency employees service each device is not a scalable solution. Rather, using a different implementation of the architecture that reuses components of the implementation for

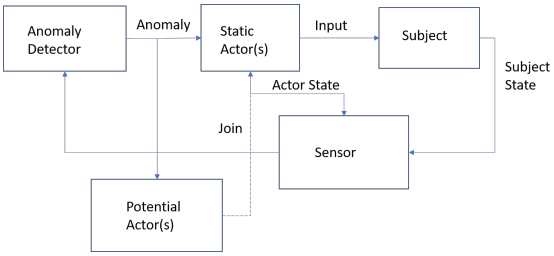

Figure 5: *A Dynamic Hybrid Loop, as described in [4]*

the driving problem, battery information from these devices could be shared with an online community, and members living near the devices could replace their batteries when they see that they are low.

Note that this type of scalability carries as a necessary sub-goal technical considerations for scalability, such as a robust server that can handle a large number of connections. This goal is easily met if the implementation interoperates with scalable technologies. For example, if the implementation needs to support instant messaging, the goal of scalability would require that the chat system be able to support a large number of clients. If the implementation interoperates with an existing, scalable system, it meets this goal and contributes to the goal of interoperability.

IV. Starting Points

IV.i. Work Flow Abstraction

Before attempting to construct an architecture that supports the driving problem and meets the goals from section III, it is helpful to be more precise about the abstract work flow the architecture is supposed to support. Fortunately such a work flow, which encompasses all the examples considered, has already been described by Dewan in [4]. This work flow, called a *Dynamic Hybrid Loop*, is shown in figure 5.

The dynamic hybrid loop is an extension of the *Closed Loop System* prevalent in control theory. Closed loops consist of an actor, a subject and a sensor. The actor acts on the subject and the sensor measures the state of the subject. Based on the information from the sensor the actor changes its behavior to drive the subject closer to some desired state. A typical example is a controllable oven. A heating unit (the actor) increases the temperature of the oven (the subject). A thermometer (the sensor) measures the temperature and tells the heating unit either to continue to heat the oven if the temperature reported by the sensor is less than the desired temperature or cease heating it if the temperature exceeds the desired temperature.

The dynamic hybrid loop makes two modifications to this model. First, it is *dynamic* in the sense that in addition to *static actors* it has *potential actors*. These are actors that are not initially a part of the loop, but can join the loop based on events received from an *anomaly detector*. The HVAC repair person as described in section II.i is an example. In the driving problem potential actors in one loop can be static actors in another. For example, students are static actors in their own loops, but since they have the ability to join a peer's loop, they are a potential actor in their peer's. The second change the dynamic hybrid loop makes to the closed loop is that it allows actors - both static and potential - to be either humans or machines. In this sense it is a *hybrid* loop. In typical closed loops, the actor is a machine. As noted in section II.i, the architecture should support these hybrid scenarios, and the abstraction of a dynamic hybrid loop in general.

IV.ii. Eclipse Helper

Systems supporting scenarios similar to the driving problem already exist; the one most closely related to the goal system is Eclipse Helper, which is described by Carter and Dewan in [5]. Eclipse Helper is a collaboration tool designed to allow instructors of computer programming courses to more efficiently provide help to struggling students. Students install an Eclipse Plugin that records events within Eclipse such as inserts, deletes and file scrolling and uses these to predict if they are in difficulty. If they are, instructors are notified and can view their workspaces, which are

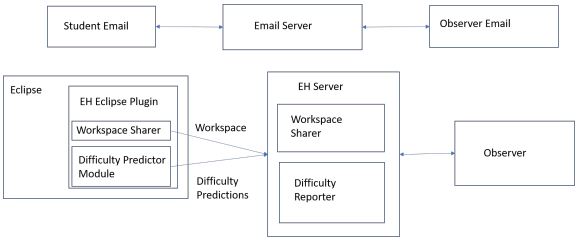

Figure 6: *Eclipse Helper Architecture. See [5] for more information.*

uploaded by the plugin to a server the instructors can access. Once the instructors determine what the issue is, they can email the students offering help. This architecture is shown in figure 6.

Eclipse Helper does not satisfy the goal of proportional reuse. It was designed to specifically support its driving problem, and as such, does not do a good job of supporting other similar scenarios. It provides no mechanism by which it can be extended, besides direct modification of its source code. So, for example, if a university had an Eclipse Plugin that automatically graded students' assignments and wanted to link this to Eclipse Helper so that instructors could provide help to students who had a low score on the assignment, this would not be possible without creating their own special version of Eclipse Helper. In addition to not being extensible, Eclipse Helper is not modular. Components of Eclipse Helper that support general collaboration are not reusable in other scenarios that need to support general collaboration. Aside from interoperating with the Eclipse Plugin Fluorite (Fluorite is described in [6]) to gather events from Eclipse and using email to facilitate communications, Eclipse Helper uses custom solutions for all other problems. Its server is custom-made, which carries implications for the goal of scalabilitly. Not only could the Eclipse Helper server likely not handle a large number of clients, but the architecture was not even designed with scalability in mind; it was designed for use by a single instructor individually assisting any student in difficulty.

Eclipse Helper is a solution for the driving problem, but it does not satisfy the goals from section III. This paper presents an architecture that does satisfy these goals as a series of modifications to the Eclipse Helper design, each bringing it closer to realizing them.

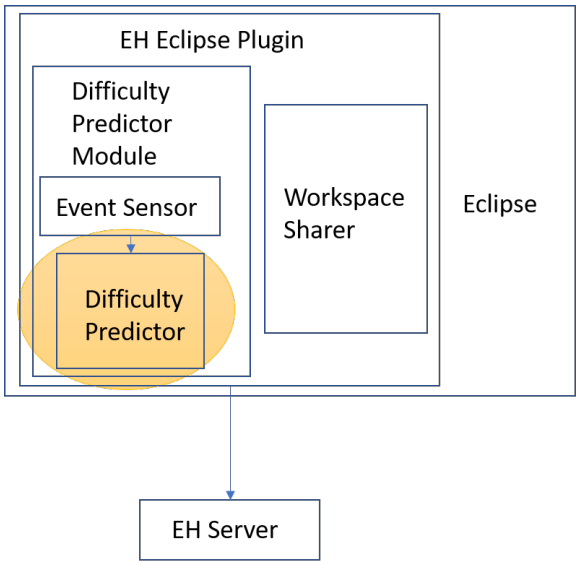

Figure 7: *How Eclipse Helper handles difficulty prediction.*

V. Morphing Eclipse Helper Into Goal System

V.i. External Processing of Sensor Data

An obvious obstacle to Eclipse Helper meeting the system goals is its tight coupling to Eclipse. For example, if one wished to extend Eclipse Helper to support an online browser, such as Ideone, all difficulty prediction logic would need to be reproduced, as it all happens within the Eclipse Helper Eclipse Plugin. The way difficulty prediction is done is shown in figure 7.

A more modular way to handle difficulty prediction is to break the difficulty predictor module into its two logical components: the event sensor and the difficulty predictor. The difficulty predictor can then be moved to the server because nothing it does is Eclipse-specific, so

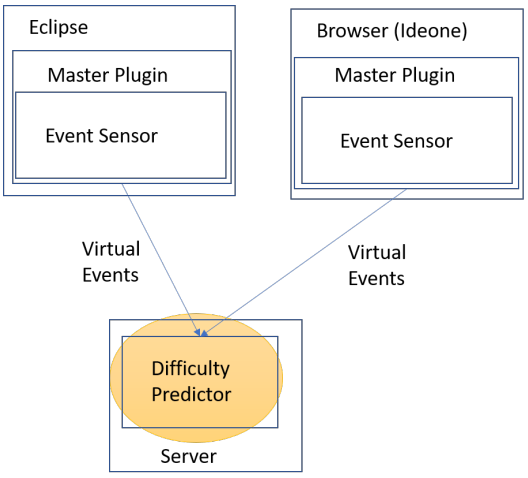

Figure 8: *A more modular way to handle difficulty prediction.*

long as it is modified to respond to virtual events rather than physical ones. The event sensor, after converting the physical events it records into virtual ones, sends them to the server, which contains the logic for using these events for difficulty prediction. This removes the need for the logic of the difficulty predictor to be reproduced in every environment where it can be used. Each environment should collect the events needed by the difficulty predictor, convert them to virtual events, and send them to the server for processing, which is where the difficulty prediction takes place. This scheme is shown in figure 8.

This is a general principle followed by the architecture: specific environments (e.g. Eclipse) should only perform operations that are specific to that environment (e.g. collecting physical events). Any function that can be shared across multiple environments (e.g. difficulty prediction) should be performed on the server to avoid duplicated logic. This implies that only simple events should be sent from the environment to the server, which is an important design principle for a different reason: it allows events to be used in ways other than originally intended, without the need to change anything on the environment side. For example, the events from the event sensor used in difficulty prediction could also be used as a rough metric of how much activity is occurring, provided the server is extensible.

V.ii. Decoupling Sensor Processing from the Server

An extensible server means the processing of sensor data happens in a process outside of the server itself. This is necessary for two reasons. As noted in section V.i, if all sensor processing happened in the server code itself, processing sensor data in a new way would require modifying the server's source code. In addition to using existing sensor data in new ways, the server also needs to be extensible so that when new sensors are added, the server can be updated to handle the data being sent by them.

A design pattern that meets this need is the *message bus*, as described in [7]. A message bus is a relay through which independent applications can communicate with each other. The specific type of message bus useful here is the *message bus with content-based publish/subscribe*. *Message bus clients* tell the message bus what messages they are interested in receiving and when the message bus receives a message from a sensor, it forwards it to all clients who have subscribed to messages of the message's type. An example is shown in figure 9.

Messages can come either from sensors or from other message bus clients. Message bus clients sending messages to each other through the message bus will be referred to as *client piping*. This allows message bus clients to take advantage of each others' processing. An example is shown in figure 10. Client piping allows message bus clients to be highly modular. The difficulty predictor, though it needs some way to communicate its predictions to the outside world, does not have to implement this functionality. It can count on another specialized message bus client, such as the notifier, to do this on its behalf. This allows for common functionality to be reused across clients. Many conceivable message bus clients need to communicate processed information to the outside world. Rather than implementing this functionality in each one, they can rely on a

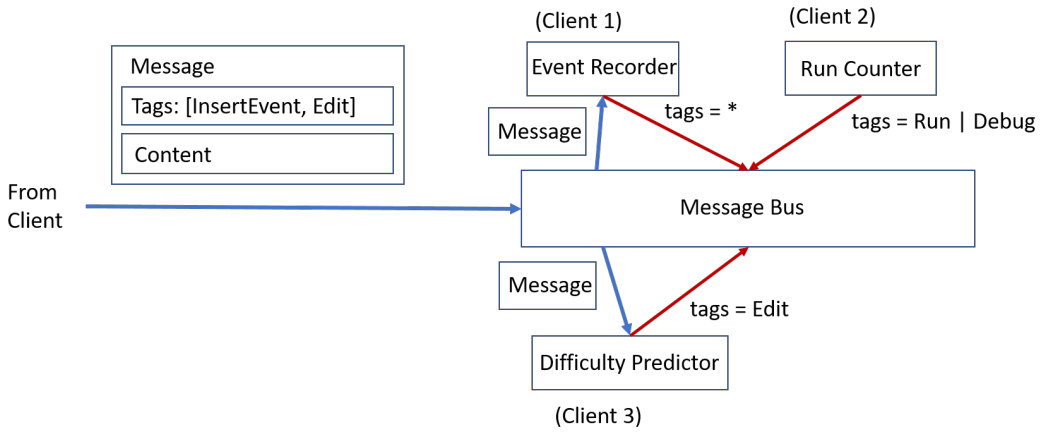

Figure 9: *A message bus with content-based publish/subscribe. In this example the message bus clients tell the message bus what messages they are interested in receiving by sending the message bus a regular expression (the red arrows). When a message arrives (blue arrow), each entry in its tags array is checked against the regular expression of each client. If there is a match, the message is forwarded to the client (blue arrow).*

specialized client to have this functionality.

V.iii. Decoupling Sensors From Master Plugin

For the same reasons that processing of sensor data should be decoupled from the server (i.e. to allow for extensibility), sensors themselves should be decoupled from the *master plugin* - the component in the students' environment with a connection to the message bus. Decoupling the sensors from the master plugin and providing a way for new sensors to be added to the system allows users to tweak the system to their custom needs by adding the appropriate sensors. Returning to the example in section IV.ii, suppose a university has a plugin that automatically grades students' assignments. With sensors decoupled from the master plugin they can connect this plugin to the master plugin and have it send grading events to the message bus.

V.iv. Identifying Universal Message Bus Clients

Although the architecture aims for flexibility and is designed to allow users to build custom solutions to their specific problems by adding

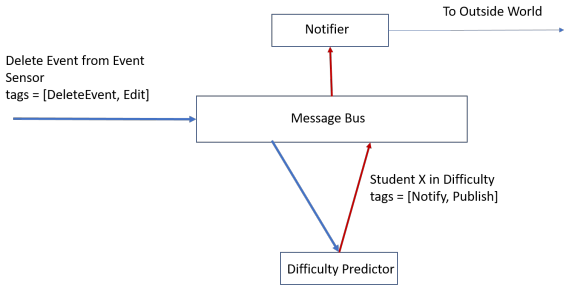

Figure 10: *An example of client piping. In this example a delete event is sent from an event sensor to the message bus. Its tags include "DeleteEvent" and "Edit". The difficulty predictor message bus client has subscribed to at least one of these message types, so the message is forwarded to it. Upon receiving the message, it determines that the student the message came from is in difficulty. It then sends a message to the message bus with the tags "Notify" and "Publish", along with a message describing the situation. This message is forwarded to any client interested in such messages. In this example, there is just one, the Notifier client, which does something like publish the message to a publicly viewable place.*

sensors and message bus clients, there are general message bus clients that are useful to almost all work flows.

The first of these is concerned with communicating the *instantaneous state* of actors to observers, which may themselves be actors in their own loops. An example of where this is useful is the fitness tracker application from section II.i. Here the instantaneous state of the actors (how many steps they have taken) is communicated to other actors (walkers) and observers (coaches, etc.). Supporting this requires a message bus client capable of disseminating information to all actors and observers in a scalable way. The client that handles this task is the *notifier client*.

A second universal message bus client is the *aggregator*. It collects information from all actors in the network and condenses it. An example that demonstrates the particular usefulness of the notifier and aggregator in the driving problem is that of having an aggregator that records how many students in a class have begun work on an assignment. Through client piping to the notifier client, this information can be shared with all students in the class. In the same way that being aware of their friends' fitness activity can encourage people to be more active, being aware of their peers' programming activity could encourage students to begin assignments early.

In addition to having a way to communicate the instantaneous state of actors, there is also a need to be able to communicate their *historical state*. In the driving problem, the historical state of students would include their current workspace as well as its state throughout time. The message bus client that allows observers to view historical state is called the *cloud store* because it needs to be a scalable file repository that is accessible from any computer and ideally most internet-connected devices; for example, smart phones. Historical state is necessary because prior to collaborating with actors, observers needs to know their current context. This requires knowing both the current state of the actors and their historical state. For example, it may be difficult to figure out how students broke their code just from looking at their current workspaces. Being able to navigate through recent changes to the workspaces makes this an easier task.

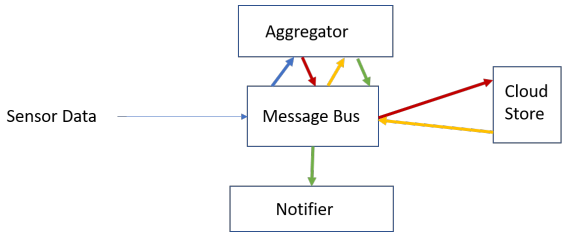

Figure 11: *Coupling instantaneous and historical state. An event from a sensor is forwarded to the aggregator. The aggregator decides it needs to publish a message through the notifier, but it first requests a link to the associated historical state from the cloud store. After receiving this link, it publishes it along with its original message through the notifier. In the driving problem this may happen if the difficulty detector client - here replacing the aggregator - detects that a student is in difficulty. It may request a link to the student's workspace history and then publish a notification with this link and the notification of difficulty through the notifier. Then interested observers can view the student's workspace (along with its history) to decide if they are able to offer assistance.*

In addition to the notifier and cloud store being universal message bus clients, their interaction is also universal. An example is shown in figure 11.

V.v. State of the Art Collaboration and Beyond: Lightweight, In-Place and Artifact-Based

The architecture now satisfies the goals as far as concerns everything except how explicit collaboration takes place between an observer and an actor; i.e. how both perform work on the same artifact simultaneously. In Eclipse Helper this happens through email, which is problematic for several reasons. It requires a student to leave the programming environment to receive help. It feels inappropriate to send an email

for a brief message such as: "concatenate two strings with a '+' rather than a ','". And finally it has been shown that communication is far more cumbersome when the involved parties are unable to point to shared artifacts to explain what they are referring to [3]. The implementation should have, and the architecture should describe, a communications system that does not require actors to leave their environment to receive assistance from observers, allows light-weight communication (e.g. instant messaging), and provides a way to perform shared editing. Such an architecture and implementation, with additional features as well, is described in **??**. A further qualification on this last point is that the shared editing must be *receiver-initiated*. This makes the collaboration process more seamless because instructors can immediately begin helping students when desired. If instructors need to request that students initiate a shared editing session, students may fumble with setting one up, and in addition to helping to resolve the original problem, instructors may need to help them solve the problem of how to begin the session.

V.vi. Complete Architecture

This concludes Eclipse Helper's transformation. To summarize, the following changes have been made: external processing of sensor data, decoupling of sensor processing from the server, decoupling of sensors from the master plugin, identification of universal message bus clients, and addition of state-of-the-art communication tools. The complete architecture is shown in figure 12.

While this architecture contributes to achieving the goal of proportional reuse, it makes no contribution as far as concerns interoperability and scalability. These are implementation goals.

VI. Implementation

The server side (the message bus and universal clients) is general - nothing about it is tied to the driving problem of programming collaboration; it is equally usable in other opportunistic collaboration scenarios. The client side (the master plugin and sensors) is implemented in Eclipse to support the driving problem. However, other client implementations that support different scenarios are given in section VII.

VI.i. Sensor Implementation

In Eclipse the master plugin is implemented as an Eclipse plugin. It interoperates with the Eclipse plugin framework to support extensible sensors. It exposes an extension point (see [9] for more information about Eclipse and plugins) that sensors can target. On startup, it queries the extension registry maintained by Eclipse to obtain a list of all declared sensors, which it subsequently initializes. Additionally, the master plugin exposes a public method that can be called by sensor plugins to send messages to the message bus. The details of this communication are covered in section VI.iv. A diagram of this implementation is shown in figure 13.

To add a new sensor, a user writes an Eclipse plugin that targets the extension point declared by the master plugin and calls the master plugin's sendMessage method when the sensor wishes to send information to the message bus.

Several sensors were implemented to support the driving problem. These include:

- Editor Contents Collector
- Console Contents Collector
- Time Tracker
- Self-Reported Difficulty Sensor

These are fairly self-explanatory. The Editor Contents Collector listens for changes to any document in the workspace and sends these changes to the message bus. The console contents collector does the same except for the console rather than for the editor. Any time a program is run within Eclipse and outputs to the console, the output is sent to the message bus (similarly for any compilation errors that are written to the console). The time tracker logs how much time students actively spend in Eclipse. If no activity is detected (activity is

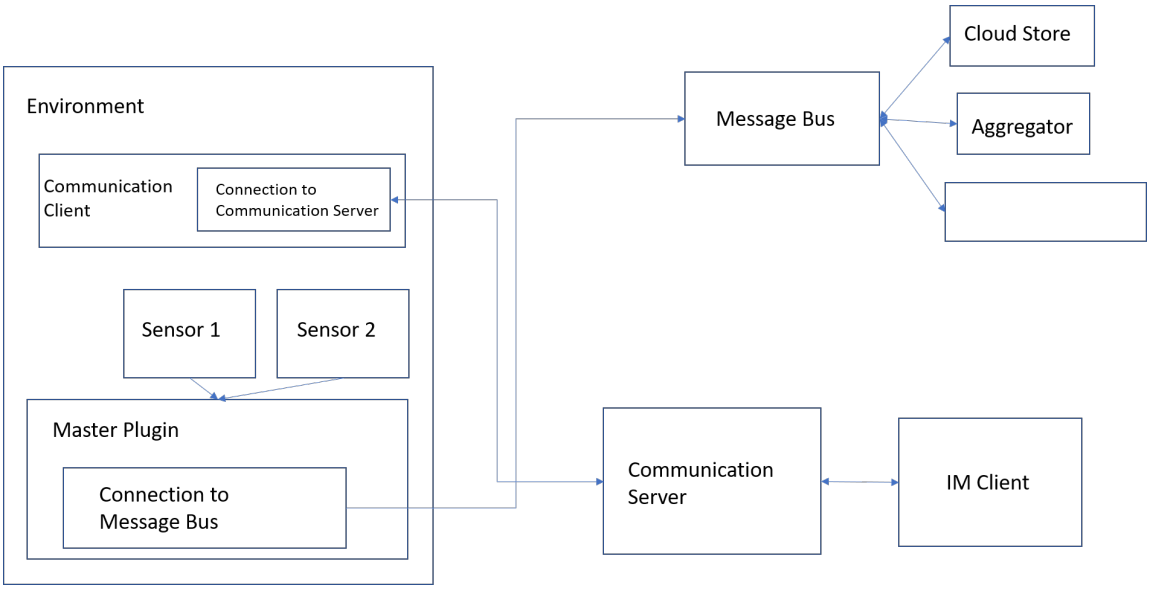

Figure 12: *Complete architecture of the goal system.*

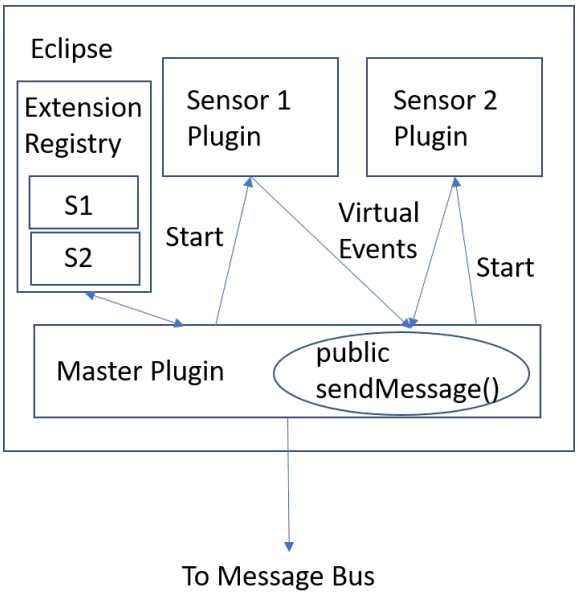

Figure 13: *Sensor implementation via Eclipse plugins.*

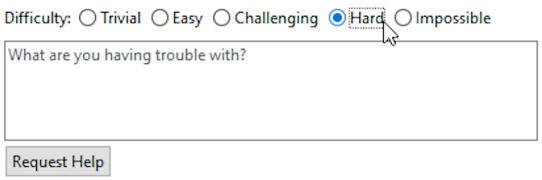

Figure 14: *Self-Reported Difficulty Sensor UI.*

detected through interoperating with Fluorite, much like Eclipse Helper) for a certain period of time, the "session" is ended and a message indicating the start time, end time and length of the session is sent to the message bus. A new session begins when activity is again detected. The self-reported difficulty sensor provides a UI to students for reporting how much difficulty they are experiencing on a scale of 1-5. When they change their level of difficulty a message is sent to the message bus. This sensor also provides a text box that students can populate with a help query to explicitly request help. Any time they click "Request Help" on a button below the query form, their request is sent to the message bus. This UI is shown in figure 14.

VI.ii. Message Bus Implementation

This section describes the implementation of a light-weight message bus with content-based publish/subscribe. A custom solution is presented rather than interoperating with an existing implementation for reasons discussed below. First, the message bus also acts as a starter. When it is initialized, it reads a file that contains a command describing how to start each message bus client (to add a new message bus client append a line to this file) and creates a process for each client and attaches a writer to its stdin and a reader to its stdout. This is how the message bus communicates with clients and how clients communicate with the message bus. Each message begins with an integer giving the number of characters in the following message - this is to avoid the use of delimiters that may occur within the message itself - followed by the message. Message bus to client communication only forwards messages. Client to message bus communication can either be to forward a message to other clients (client piping), to subscribe to specific types of messages (which is done by sending a regular expression that will match one of the tags of messages the client is interested in receiving), or to write to the message bus's output. To declare what type of message the client is sending to the message bus it prefaces the message with one of three starting strings: "<msg>", "<tags>" or "<output>". The implementation includes libraries for java, node and python that implement this protocol so writing clients in any of these languages is especially easy. The protocol is simple enough that writing a custom client in any language should be simple. This is one of the benefits of a custom message bus implementation: it allows users to write clients in any language, as every language has support for standard I/O, whereas more complicated protocols, such as sockets, may not have uniform implementations in all languages. Similarly, users' familiarity with standard I/O lowers the barrier to creating message bus clients. A diagram illustrating this implementation is given in figure 15.

VI.iii. Human Communication Implementation

To implement the architecture design of lightweight, in-place and artifact-based collaboration, as well as to satisfy the goals of scalability and interoperability, the implementation interoperates with the Eclipse Communications Framework (ECF) and the Extensible Message Passing Protocol (XMPP). ECF is a suite of communications tools for Eclipse. For more about working with ECF see [10]. It provides instant messaging and shared editing, though not receiver-initiated shared editing, which is a new concept and other collaboration tools that offer shared editing do not offer receiver-initiated shared editing either, presumably for privacy and security reasons. XMPP is an extensible messaging protocol with "software for every platform and libraries for every language" [11]. XMPP is also one of the main protocols supported by ECF. Additionally it claims to be ideal for IoT applications [11], which will be discussed in section VII.ii. Setting up the communications system involves running an XMPP server - of which many implementations exist; a popular open source version is Openfire - and creating connections to the server on the actor's and observer's end. The actor's connection is established through a modified (to allow for receiver-initiated sharing) version of ECF and the observer's end can either be through ECF (if the instructor is using Eclipse) or through any number of XMPP clients. Many client implementations exist, both for desktop and mobile devices. One popular desktop implementation is Spark. This is shown in figure 16.

As specified in the architecture, this implementation allows for students to communicate within the programming environment and supports both instant messaging and receiver-initiated sharing. Further, both ECF and XMPP have demonstrated their ability to support internet-scale loads.

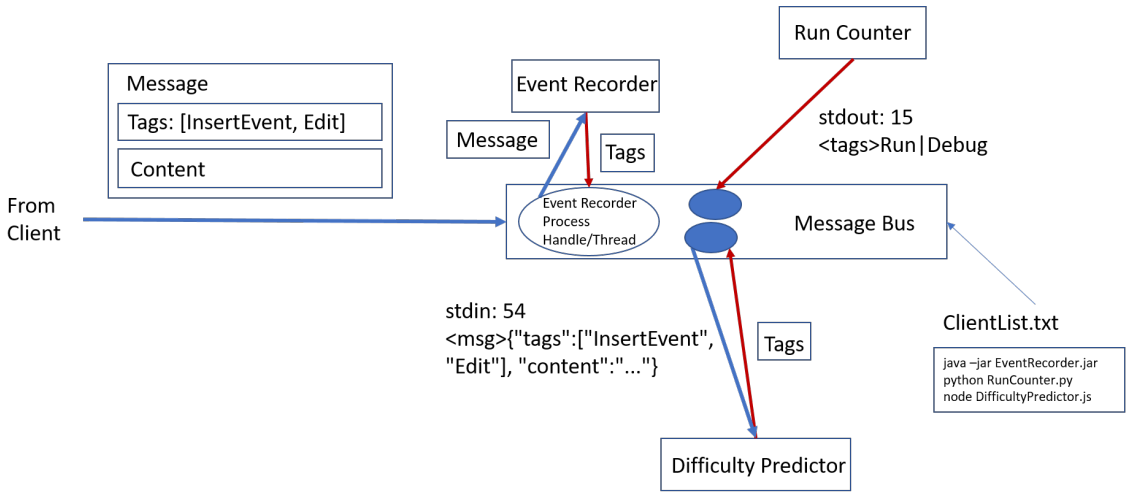

Figure 15: *Message bus with content-based publish/subscribe implementation.*

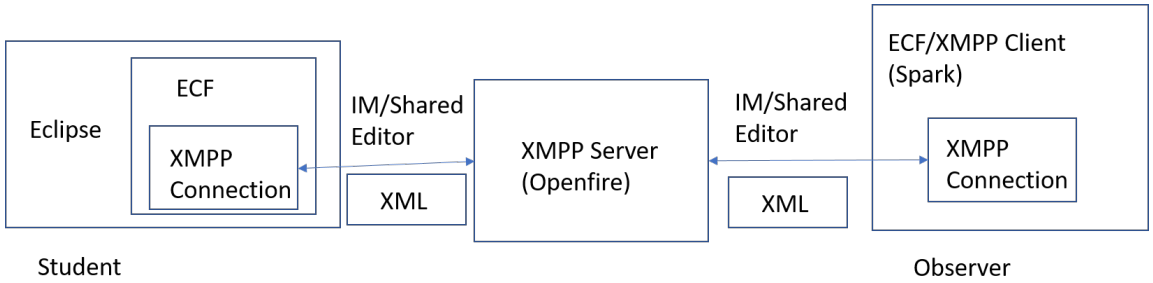

Figure 16: *Human-to-human Communication with XMPP and ECF.*

VI.iv. Master Plugin to Message Bus Communication

In addition to supporting communication for explicit collaboration, the implementation needs to provide a way for sensor data to be sent from the master plugin to the message bus. This can also be achieved using XMPP. Thanks to the communications setup described in section VI.iii, there is already an XMPP server running. By creating an XMPP connection in the master plugin that can be accessed by sensors through the master's exposed method for sending messages to the message bus and on the message bus, sensors can send messages to the message bus. XMPP has client libraries in many languages, so this poses no difficulty. This satisfies both the interoperability goal as well as the scalability goal. As already noted, XMPP servers are designed to handle a large number of clients and high traffic among them. This implementation has the added benefit that it only requires the running of a single server for both human-to-human communication and master to message bus communication. A diagram of this implementaiton is shown in figure 17.

VI.v. Cloud Store with History Implementation

The cloud store client needs to be a file repository that stores the history of each file and has a way to easily present both the file and its history to observers. Ideally it is accessible from a number of devices. Google Drive perfectly fits this description, and in addition is easy to interoperate with. Google Drive has a public API that allows for the programmatic creation and editing of files, specifically Google Docs. Google Docs internally store a list of all revisions made to them, so by simply modifying the Google Doc, the history of the file is automatically stored. This history can be viewed within the Google Doc itself or in more detailed ways through third-party extensions such as DraftBack. Implementing the cloud store is as easy as interoperating with Google Drive, which is simple thanks to its API and client libraries. A diagram is shown in figure 18.

Sharing files through Google Drive is scalable and also offers security and privacy, which has been shown to be an important feature in collaborative systems ([5]). On the students' side, there is an option within the Editor Contents Collector plugin to set the privacy of the files that are uploaded to Google Drive.

VI.vi. Notifier Implementation

The notifier client needs to be able to disseminate information to a large number of observers and ideally in a medium that can be accessed from a variety of devices. Facebook provides a perfect solution. The timeline concept is ideal for this type of communication, it is scalable, offers privacy and security, is viewable on many platforms and has a well-documented and easy to use API. The notifier client receives messages from the message bus - either text or images - and posts them to the wall of a Facebook page that is set up on the first run of the client. This Facebook page is viewable to all observers.

Both this implementation and the Cloud Drive implementation support the coupling described in section V.iv; a post can be made to the Facebook wall containing a link to the workspace folder in Google Drive for a specific student.

VI.vii. Complete Implementation

This completes the implementation of the architecture. It includes implementations for sensors and the master plugin, the message bus, human-to-human communication, master plugin-to-message bus communication and two universal clients. As noted in the discussion of the architecture, the server side - the message bus along with XMPP software - is reusable among all opportunistic collaboration scenarios. This implementation is shown in figure 19.

The Eclipse implementation should only contain components that are specific to Eclipse,

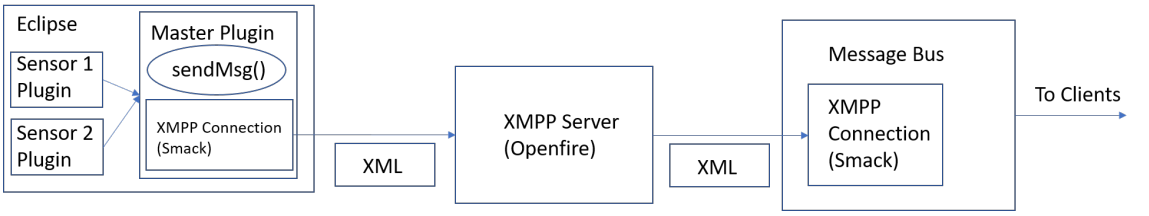

Figure 17: *Master plugin to message bus implementation using XMPP and the XMPP server for human-to-human communication.*

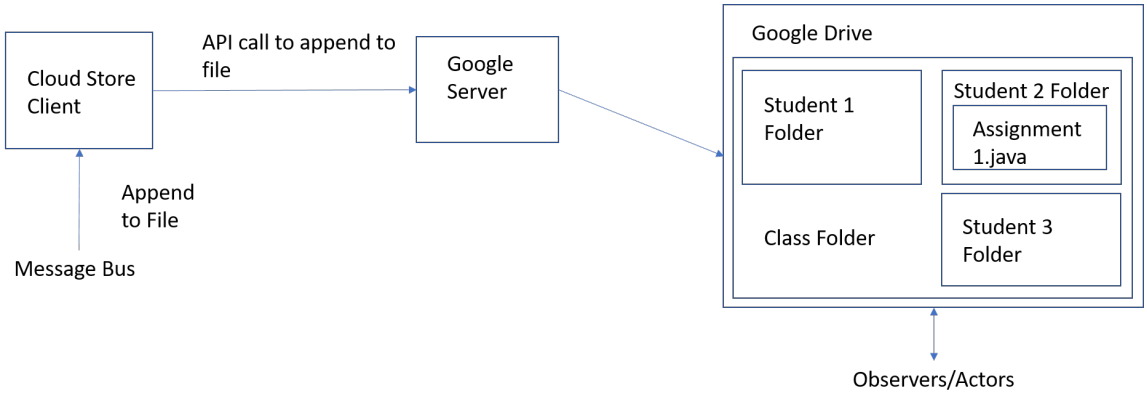

Figure 18: *Cloud Drive Implementation by Google Drive.*

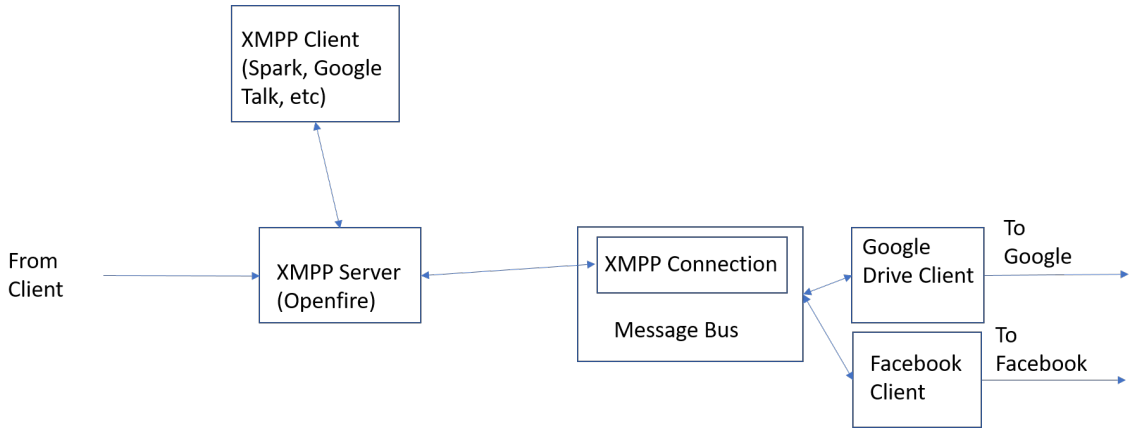

Figure 19: *Complete server side implementation. Message bus + XMPP.*

and will necessarily have different implementations in different environments (e.g. the way in which the editor contents are collected is necessarily an environment-specific implementation). This implementation is shown in figure 20. It includes the Console Listener, Editor Listener, Time Tracker and Difficulty Reporter, as discussed in section VI.i.

VII. Implementations for Different Scenarios

To demonstrate the flexibility of the architecture and implementation given, the following sections present implementations that support opportunistic collaborations scenarios that differ from the driving problem. As already noted, the server side (XMPP server, message bus and message bus clients) remains unchanged. Creating a new implementation, in both examples given here and most conceivable ones, amounts to creating an XMPP connection in the client environment, exposing a way for sensors to send messages through this connection, and writing sensors.

VII.i. Online IDEs

The implementation presented in this section also addresses the driving problem, except it is for a Chrome extension to support online IDEs, rather than for an Eclipse Plugin to support Eclipse development. The extension creates an XMPP connection using the javascript library Strophe. It collects editor contents by scraping the DOM and sends them to the server using the same XMPP messages that the Eclipse implementation uses. Without changing anything on the server side, the file history is stored in a Google Doc in the same manner it is in the Eclipse implementation.

VII.ii. Microcontrollers

As discussed previously, this architecture is especially useful in IoT applications. As such, this section gives an implementation for the Raspberry Pi. The server's message bus is duplicated on the Pi to create an XMPP connection and to allow for extensible sensors, which are the message bus clients. The microcontroller has a button programmed to append a line to a Google doc each time it is pressed to demonstrate that with this implementation data can be collected from a physical environment and published to many observers. Additionally it has an LED that can be controlled by incoming XMPP messages to demonstrate that the physical environment can be affected by remote observers.

VIII. Evaluation

VIII.i. Proportional Reuse

To properly evaluate how well the architecture and implementation satisfy the goal of proportional reuse requires creating implementations that solve related problems and looking at the effort required to do so, as well as the amount of the original solution that can be reused. Implementing solutions for more distantly related problems should require more effort than implementing solutions for more closely related problems, as closely related problems should be able to reuse more components of the original solution than distantly related problems. Of course, the complexity of the problem will also affect how much effort is required. Three scenarios are considered:

- Adding a sensor to support automatic grading
- Supporting online IDEs
- IoT Framework

Adding a sensor to support automatic grading involves adding a new sensor to Eclipse and adding a client to the message bus to process events from this sensor.

Supporting an online IDE involves implementing a communications system and a master plugin that can send messages to the message bus. Additionally any desired sensors will need to be implemented. Fortunately any processing of sensor data will remain the same.

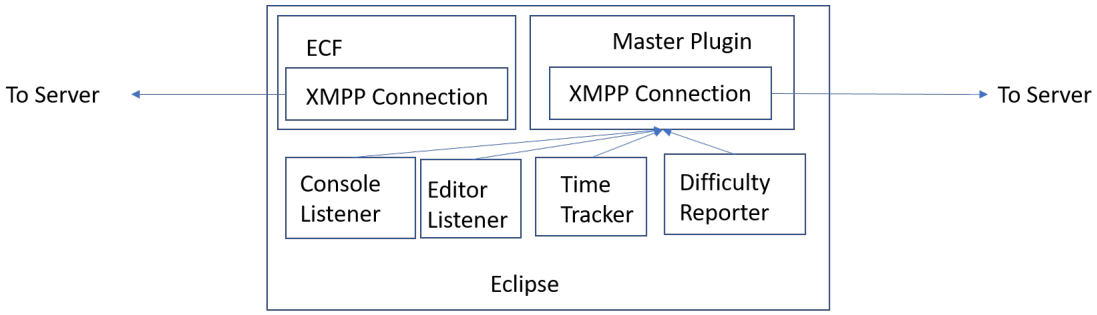

Figure 20: *Complete Eclipse implementation.*

So if a programming difficulty detector exists as a message bus client, the online programming environment can make use of it just as the Eclipse implementation would.

The IoT Framework will involve the same work that supporting the online IDE did, except it will not be able to take advantage of programming-specific message bus clients like the difficulty predictor. Because of this, there will be less reuse of the original implementation, as was expected since the application is more removed. Still, a large amount of the infrasture can be reused.

These examples demonstrate that the architecture and implementation achieve the goal of proportional reuse. Figure 21 provides a summary of this discussion.

VIII.ii. Scalability

Scalability issues have been passed off to XMPP implementations, Google Drive, and Facebook, all of which have demonstrated their robustness under large loads. Additionally, the implementation allows for anyone on the internet to collaborate with anyone else on the internet, which was the motivating goal. Because messages are sent through XMPP, which requires that students are logged into an XMPP account, the problem of authentication is also solved.

VIII.iii. Interoperability

The given implementation interoperates with Google, Facebook, Openfire, Smack (an XMPP library for java), Spark and Eclipse. The only place where it fails to take advantage of an opportunity to interoperate with existing technology is with the custom message bus implementation, although the reason for creating a custom solution is motivated in section VI.ii.

IX. Discussion

IX.i. Contributions

This paper has given an architecture and implementation that support scalable mixed-activity opportunistic collaboration.

The architecture makes a couple of significant contributions. It puts forth the idea of the importance of coupling instantaneous and historical state information as well as creating the notion of receiver-initiated sharing.

The implementation supports the driving problems and meets the set out goals. Additionally it is able to facilitate both human-to-human and master-to-message bus communication with a single server, an idea that can certainly be translated to other domains.

The message bus implementation presented here is of potential interest in any application where a message bus is needed. And in fact is used in the microcontroller implementation in section VII.ii.

Both Google Docs and Facebook were used in novel ways. Machine-generated content was automatically uploaded to Google Docs, and a machine layer was placed between the humans in the human-to-human network that Facebook was designed to support. And in the microcontroller scenario, Facebook can be used as a

Implementations	XMPP Server	XMPP Client	Message Bus	Cloud Store (Google Drive)	Notifier (Facebook)	Programming Clients (e.g. Difficulty Predictor)	ECF	Master Plugin	Sensors
Adding Sensor	✓	✓	✓	✓	✓	✓	✓	✓	✓
Online IDE	✓	✓	✓	✓	✓	✓	×	×	×
IoT	✓	✓	✓	✓	✓	×	×	×	×

Figure 21: *Proportional Reuse Evaluation.*

Machine-to-Human network.

IX.ii. Future Work

Immediate future work will revolve around implementing more message bus clients, such as a time tracking client to aggregate the information recorded by the time tracking sensor and publish it to Facebook and a difficulty reporter that aggregates the information recorded by the difficulty sensor and publishes it. The system needs to be tested for robustness, and once verified, deployed for a field study.

Longer term work includes comparing this implementation to other non-scalable UIs, creating implementations for other scenarios (e.g. helping students in English class write papers - see [12]), creating a comprehensive IoT Framework, making server setup simple with configuration management, and making a running server easy to manage by creating a web tool to configure message bus clients.

X. Acknowledgement

The author would like to thank Prasun Dewan for his many helpful discussions about this architecture and its implementation, as well as the structure of this paper, as well as David Stotts for his feedback on early versions of this paper.

References

[1] Hollan, J. and S. Stornetta. *Beyond Being There*. in ACM CHI Proceedings. 1992.

[2] Dourish, P. and V. Belloti. *Awareness and Coordination in a Shared Workspace*. in Proc. ACM CSCW'92. 1992.

[3] Gutwin, C. and S. Greenberg, *A Descriptive Framework of Workspace Awareness for Real-Time Groupware*. Comput. Supported Coop. Work, 2002. 11(3): p. 411-446.

[4] Dewan, P. *Inferred, "Intelligent" Awareness to Support Mixed-Activity Collaboration*. IEEE CIC 2016.

[5] Carter, J. and P. Dewan. *Mining Programming Activity to Promote Help*. in Proc. ECSCW. 2015. Oslo: Springer.

[6] YoungSeok Yoon and Brad A. Myers. *Capturing and Analyzing Low-Level Events from the Code Editor*. Workshop on Evaluation and Usability of Programming Languages and Tools at SPLASH 2011 (PLATEAU 2011). October 24, 2011. Portland, Oregon, USA.

[7] Reiss, S. P. (1990). *Connecting Tools Using Message Passing in the Field Environment*. IEEE Software 7(4): 57-66.

[8] Hegde, R. and P. Dewan. *Connecting Programming Environments to Support Ad-Hoc Collaboration*. in Proc 23rd IEEE/ACM Conference on Automated Software Engineering. 2008. L'Aquila Italy: IEEE/ACM.

[9] *Eclipse Platform Technical Overview*. International Business Machines Corp. 2006.

[10] Fabio Calefato and Mario Scalas, *Adopting the Eclipse Communication Framework: The Case of eConference*. CEUR Workshop Proceedings. Volume 436. 2008.

[11] XMPP - eXtensible Messaging and Presence Protocol, http://xmpp.org

[12] Long, D., *Mixed-focus Difficulty-Triggered Collaborative Writing: Interoperable Architecture, Implementation, and Evaluation,*

in Computer Science. 2016, University of North Carolina at Chapel Hill.