Coordinating Collaborative Interactions in Web-based Mobile Applications

Kennedy Kambona, Lode Hoste, Elisa Gonzalez Boix & Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
{kkambona, lhoste, egonzale, wdmeuter} @vub.ac.be

ABSTRACT

Mobile applications for interactive surfaces that utilize the web as a platform now have the ability to provide richer interactions hitherto unrealized by running them on isolated devices. These modern applications can now support proximal and remote collaborative interactions for multiple clients simultaneously connected to each other. Most technologies however currently lack programming language abstractions for coordinating complex interactions, such as to define, detect and combine complex events coming from multiple clients or other software entities. Furthermore, they lack the expressiveness required to support non-trivial levels of collaborative interactions for connected clients.

In this paper we identify two software mechanisms that web-based mobile applications should provide to support the development of collaborative interactions: distributed event composition and group coordination. We present the Mingo framework, which provides dedicated coordination programmer constructs for these two mechanisms by blending techniques common in complex event processing and group communication. Consequently, we validate our framework by implementing a mobile drawing application with support for collaborative interactions and evaluate it by comparing it with a related implementation.

Author Keywords

Collaborative mobile applications; coordination; complex event processing; interactive applications

ACM Classification Keywords

H.5.3 [Group and Organization Interfaces]: Collaborative Computing, Computer-supported cooperative work, Web-based interaction

INTRODUCTION

Traditional software applications have recently started experiencing a fundamental shift due to cheaper broadband internet connections. Many applications are increasingly residing on the web and their services provided to clients (e.g. mobile phones) using a multitenant cloud-based architecture. These applications can now run in vastly improved resources compared to those available a decade ago, and have the ability to provide richer functional interactions.

In this paper we investigate how to provide software support for collaborative surface interactions using such contemporary infrastructures. Coordination of interactions is vital in applications running on the because it is involves managing multiple users, tasks and devices. This is becoming increasingly important as evidenced by the popularity of ubiquitous devices with internet access. Recent advancements in standardized web-based technologies such as HTML5 have increased the support required for coordinating various types of collaborative applications, such as document editing software, instant messaging and online multiplayer games.

The common technique employed by existing applications, however, is to impose restrictions on software entities in order to deal with the complexities of simultaneous collaborative interactions of participants within the same session. An example of such restrictions is to enforce exclusive access, where interactions on an object is allowed for only one participant at any one time. In a collaborative drawing application, for instance, this means that a participant is said to exclusively 'own' shape (e.g. by selecting and editing it), and it can only be modified by that user until the ownership is relinquished (e.g. by deselecting it). Another type of limitation is sequential access, where interactions on an object by several participants are performed in succession – in the order that they were detected. Rather than imposing such software restrictions, we envision a programming model where events coming from a group of different participants in interactive devices can be composed and integrated. This will result in a mechanism in which advanced collaborative interactions can be orchestrated, a significant advantage in mixed interactive surface applications.

We propose the Mingo framework, which provides programming abstractions for coordinating collaborative interactions in mobile applications running over the web. Mingo combines techniques from complex event processing and group coordination to assist developers implement applications supporting collaborative interactions such as collaborative drag-

ITS 2015, November 15–18, 2015, Funchal/Madeira, Portugal.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3899-8/15/11…$15.00.
DOI: http://dx.doi.org/10.1145/2817721.2817742

ging on a surface. Mingo has been implemented as an extension to JavaScript, the dominant language on the web. Interactions are modelled in the asynchronous message-passing style which aligns with JavaScript's event-driven programming model. Mingo constructs thus allow developers to specify collaborative interactions involving several participants in a declarative and reusable way. Mingo is targeted for devices that are co-located or in low-latency environments to provide immediate feedback for the interactions.

We present our model in the subsequent sections. We first motivate the need for coordinating mobile applications on the web and present some related concepts. We then present our coordination model, the Mingo framework and its coordination constructs. Using Mingo, we proceed to implement a collaborative drawing editor and compare it with a similar implementation using the current state-of-the-art. We conclude by discussing possible future improvements to the framework.

THE NEED FOR COORDINATING COMPLEX COLLABORATIVE INTERACTIONS

A number of research has identified the need for collaborative interactions – user actions spanning a distributed system that occur together – such as synchronous gestures [9] and co-browsing [2]. This allows users in proximity to combine multiple devices heterogeneously to exploit pooling input capabilities, content and resources such as displays.

In this paper, we motivate the need for providing programming support for such advanced coordination mechanisms for complex interactions. We introduce a running example of a sample application requiring these mechanisms: an illustrative collaborative drawing editor.

Motivating Example: Online Collaborative Drawing Editor

Consider a traditional collaborative drawing application similar to Cacoo[1] where several distributed participants on mobile phones are connected via the web using a server. The participants share the same canvas on multiple mobile devices. The users are participating in the same session and can therefore interact at the same time. The shared canvas can be used by the several participants connected via the web to draw and interact with (or manipulate) shapes already drawn on the canvas.

In addition to the normal functionality provided by a traditional drawing application (as described thus far), we envision that the collaborative web-based drawing application further allows users on mobile devices to perform advanced collaborative interactions to users[2]. We illustrate two examples of these collaborative interactions that can be applied:

(a) Collaborative resize interaction. Consider when one participant on a mobile device starts to drag a drawn shape to the right, while at the same time a different participant in the same session drags the same shape to the left (Figure 1a). Typically an application recognizes two separate sequential interactions (e.g. drag-right and drag-left respectively), and proceeds to enforce techniques to determine ownership and which action to apply. In contrast, our sample application recognizes this as a single collaborative resize interaction. As a result, the application then reacts by causing the shape to resize according to the input of the participants.

(b) Collaborative bind interaction. Consider when one participant drags onto an empty canvas to the right, while another one drags on the canvas to the left. When both participants reach the canvas borders as shown in Figure 1(b)[3], an application could then virtually bind the canvases in the two devices to give the impression of an extended shared canvas. Participants can then for instance share objects by 'throwing' them to other devices using a sliding gesture.

Similar features can be applied to other applications in need of collaborative interactions: for instance in a virtual dancing game, where two or more users are required to press a sequence of button combinations or other actions at the same time to successfully perform a cooperatively-synchronized dance move in a group.

Requirements

We now identify two requirements that support coordination of complex interactions for collaborative web-based mobile applications.

i) Processing and Composition of Events:
Interactions in most mobile web applications are primarily internally represented as events. A basic programming challenge in this setting is that it is arduous to detect and distinguish between simple events that the application receives e.g. a local drag operation, from the composed collaborative events e.g. the collaborative resize from two or more participants in a session. A simple event involves one participant and is usually local. The collaborative events can possibly involve several distinct participants. Our vision is that developers of such collaborative applications need to be able capture to both types of events.

Detecting simple (or *local*) events from different areas of the application and subsequently creating a composed collaborative event is known as *event composition*. Programmers face great difficulties (denoted as accidental complexity in [17]) when specifying and orchestrating such interactions or events. Interactive applications developed using various

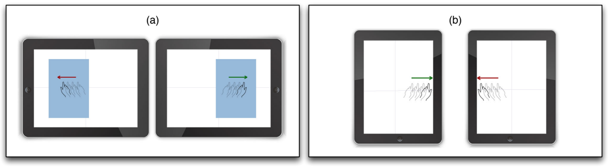


Figure 1. Collaborative interactions for a drawing application: (a) collaborative resize (b) collaborative bind

[1]Cacoo - Create diagrams online, `https://www.cacoo.com`
[2]The users should already be aware of this e.g. from a description of the application features

[3]The right border for the first participant and the left border for the second.

frameworks should be able to define and detect when such composed events occur.

ii) Group Coordination:
In traditional mobile and desktop applications, software entities (like shapes objects in a drawing editor, or dancing avatars in a dancing game) were designed to be implemented and used autonomously in a local device. In contrast, distributed applications using the web have the potential to run multiple instances of software entities distributed over several mobile clients, while at the same time exposing the more advanced functionality. These replicas or groups of software entities require a group coordination mechanism in order to address them as one unit and to manage their respective objects with the aim of making their operations resilient. For instance, in a collaborative drawing application, a drag event performed on a shape by a participant should be transparently propagated to all the different mobile clients of the same session that have this shape; in a dancing game a dancing avatar's local movements should be updated in all other participants' screens. The coordination abstraction should provide a means to manage the group behaviour of these replicas.

Summary
This paper envisions a programming framework that integrates these two requirements to coordinate collaborative web-based mobile applications. Developers can utilize abstractions that allow several participants in a session to interact simultaneously and collaboratively, utilizing more advanced combined distributed interactions.

RELATED WORK

Coordination Models
Most coordination models and languages for concurrent and distributed systems are based on the *tuple space* model of Linda [6]. A tuple space is a globally shared virtual data structure which allows processes to communicate by posting and reading data objects known as *tuples*. Some tuple space implementations have targeted the web such as WWWinda[7]. This approach employs a separate browser architecture with tuple spaces shared across web applications. Conceptually, a tuple represents local interactions or events. As such, developers using tuples for coordination still need to manually apply additional logic in applications in order to manage complex collaborative interactions from individual tuples.

Multi-device Interaction
Groupware frameworks such as GroupKit [15] and Sync [11] expose extensible high-level architecture or interfaces that applications are required to adhere to in order to receive support for collaboration. Sync embeds generic collaboration capabilities into several base classes that need to be extended by the application. Groupkit requires an application to extend a registration client to create conferences that act as a mediator between the application and the framework. Our vision is to have no such restrictions on applications or objects in need of coordination support.

PointRight [10] provides mechanisms for redirecting pointer input across screens in an interactive workspace. For coordination, it uses a custom blackboard architecture derived from tuple spaces which only allows a user to control a single display at a time. ARIS [3] is an interactive window space manager that uses Gaia middleware. Gaia supports management of distributed devices and services through an information repository for entities. It however provides little or no support for managing filtering and historical data context processing, which is useful for coordinating more complex interactions. Work in [9] allows support for multi-person or multi-display interaction. The driving technique behind this approach is a predetermined algorithm that uses various inputs to recognize synchronous spikes in data. However, the approach is rigid and does not support dynamic addition of gesture definitions at runtime.

Web-based Collaboration
The web was primarily intended for single-person use in its conception: the original architecture did not support real-time synchronous interaction. Nowadays, however, support of real-time interaction is increasingly popular with the advent of web technologies supporting real-time collaboration. Etherpad[4] and GoogleDocs[5] allow users to collaborate on text documents simultaneously. Schmid et al. [16] coordinate simultaneous collaborative web browsing by using specialized proxy servers that intercept web page requests and inject JavaScript code into the pages, providing functionality for simultaneous access into web pages. However, their simplified approach of interaction of multiple pointers on the screen is limited to single-pointer interaction per object. Furthermore the use of a proxy increases the *a priori* knowledge and configuration needed.

Complex Event Processing
Complex event processing (CEP) is a technique that provides support for detecting, analysing and composing multiple simple interactions or events into more meaningful or higher-level (complex) events. *Computation-oriented* CEP engines focus on aggregating data from a number of incoming events (e.g. Borealis [1]), while *Detection-oriented* CEP engines focus on finding patterns within multiple event instances rather than the aggregation of these instances (e.g. Drools [13] and Midas [17]). Computation-oriented engines are unable to detect and compose events through specific, fine-grained coordination interaction patterns, making detection-oriented engines is better suited for composing collaborative interactions. One of the most representative of detection-oriented CEP engines is Drools [13]. However, Drools lacks constructs for spatial and temporal operators that can enable us to easily express interaction patterns, significant when performing composition of interactions. Nools[6] is a rule engine for the web written in JavaScript. Both implementations are based on the Rete algorithm [5], which employs an efficient implementation for pattern-matching complex events.

MINGO
We now present our framework for coordination of complex interactions in collaborative mobile applications running on

[4]`http://etherpad.org`
[5]`http://google.com/docs/about`
[6]`http://github.com/C2FO/nools`

the web. Mingo is an object-oriented framework employing web-based infrastructure to coordinate collaborative interactions found in interactive surfaces. JavaScript has emerged as the de-facto language for web development due to the native support of all modern web environments. The framework therefore extends the language with blended constructs for group coordination and complex event processing.

In particular, Mingo employs the CEP Midas engine [17] at the heart of its execution model. Midas is a detection-oriented rule-based CEP framework for the recognition of complex interactions that allows developers to easily express advanced interaction patterns. Consequently, Midas provides ample support for modelling multi-user interactions as events. A benefit of using the Midas engine is therefore that there is no difference in reasoning over simple events (represented as facts) vis-a-vis composite or complex events. This provides significant support for advanced software engineering abstractions [17]. The engine however lacks native support for distribution on the web. We therefore extended the engine to support this, required for such collaborative interactions. The Mingo framework is mainly targeted for co-located devices and similar low-latency environments for the prompt feedback of the possible activated rules and interactions.

In addition, the abstractions supporting coordination of collaborative interactions in Mingo are designed along the principles of *separation of concerns* [12]. The main benefit that this separation provides is that it enables developers to concentrate on an application's functionality separate from the coordination mechanisms that it may require. Papadopoulos et al. [12] already identified this as a significant design principle that helps in alleviating issues related to compositionality and extensibility in the development of distributed applications.

Mingo Execution Model

The execution model of Mingo utilizes a number of technologies to achieve its coordination functionality, as illustrated in Figure 2. The server manages the coordination mechanism of the framework. It consists of the Mingo server later and the Midas CEP engine, with the Mingo bridge acting as an interface between them. The bridge translates Mingo constructs received from the clients into rules and facts to be consumed by Midas (and vice-versa).

A number of interactive client devices can be connected to the server. The clients run JavaScript code augmented with Mingo constructs necessary for coordination. When the constructs are applied in the application the Mingo runtime takes care of packaging and sending this information to the server. The Mingo layer on the server side will take care of transforming and injecting the necessary rules in the Midas engine, as represented in the constructs. Whenever a simple event that has been denoted to comprise a complex interaction is performed, Mingo intercepts its invocation and sends the event information to the server. If a complex interaction has been realised, Mingo propagates this new information to all clients in the session.

To clarify Mingo's coordination orchestration, we illustrate how a collaborative interaction is realized.

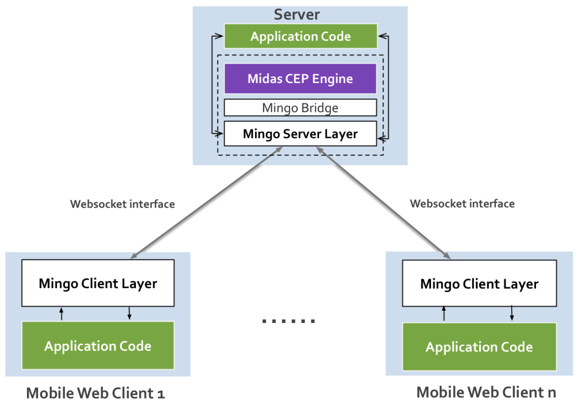


Figure 2. Mingo Execution Model

1. A developer writes the application in JavaScript and employs Mingo Client constructs to coordinate the application's distributed interactions (described in the next section).
2. The Mingo runtime sends the interactions and their relevant information to the server by means of websockets.
3. The Mingo layer on the server assembles the local interactions from client devices with their respective information and sends them to the Midas CEP engine by mapping invocations and their definitions into facts and rules that Midas can consume. Thus in Mingo a fact is an internal representation of a simple interaction in the application.
4. Midas then asserts the interactions. If the interactions trigger a pre-defined rule created from constructs previously defined by the developer, it can combine the simple interactions into one composed event.
5. The composed event is then pushed back to the runtime which in turn delivers it to the awaiting client devices that defined a handler for the composed event.

Programming Support for Coordinating Complex Interactions

In this section we describe the programming language constructs that Mingo avails for complex event composition and group coordination. Although we employ our running example in explaining these abstractions, they can also be applied to other application scenarios requiring these coordination mechanisms.

Group Coordination Abstraction.

Mingo provides the ability to coordinate the behaviour of distributed software entities by means of the `groupObject` construct. When `groupObject` is applied to a new object, Mingo proceeds to assign a group identifier to the object, and transparently assigns the same group identifier to the replicas in every client mobile device (Figure 3). The effect of this is Mingo will subsequently replicate any specified actions performed on that object to each replica of the object in the other client devices through assertions to the CEP engine. Therefore the group object can be visualized as a virtual object that

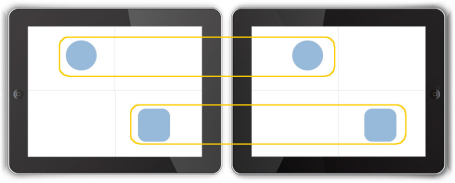


Figure 3. Two distributed object groups

links all replicated objects to it and will distribute interactions or events to the all objects with the same identifier in the same session. In our running example, we have employed the `groupObject` construct to model the shapes drawn in the shared canvas as shown in Listing 1.

Listing 1. Defining a Group Shape in Mingo

```
1  var shape = new Shape({
2    touchMove: function(e){
3      //update shape position
4    }
5    //...
6  });
7  var groupShape = Mingo.groupObject(shape, {
8    collabResize: Mingo.rule(/*rule definition here*/)
9  });
```

As shown in line 1, a developer defines a shape with basic operations e.g. `touchMove` for a mobile client (equivalent to a `mouseMove` operation on a desktop device). Additionally, in line 7 we define that shape as a Mingo group object using the `groupObject` construct. The construct takes as the first argument the shape, and the second argument an object containing the behaviour which will be part of collaborative interactions in form of a rule (explained in the next section). As shown in Listing 1, when the end user draws a shape, the Mingo creates a `groupShape` object, and the canvases of the rest of the participants in the drawing session will be populated with a replica of this shape.

The `groupShape` object conceptually represents the replicas of a shape which all belong to one group. The developer can then replicate operations through the Mingo `replicateAction` construct available on a `groupObject`. For instance, when the `touchMove` operation (line 2) is performed on any member of `groupShape`, this action will be replicated on all the other shape replicas in the mobile application in the same session.

Listing 2. Replicating Actions on an Object Group

```
1  groupShape.replicateAction('touchMove');
```

Thus Mingo group coordination constructs address a number of distributed object *replicas* as a single unit. Given a set of group objects, when a client changes a copy of the object Mingo transparently propagates updates to all the replicas in the application. An invocation on `touchMove` sends the information to the server and which is then asserted in Midas. The assertion of an replicated action triggers Mingo to systematically send the action to the rest of the users in the same session. Each replicated action is internally represented in Mingo as a rule that triggers upon the assertion related facts.

In the next section we explain how a developer can perform distributed actions such as the collaborative resize operation in line 8 of Listing 1.

Distributed Event Composition.

In web-based interactive applications, simple events are usually detected by registering a handler to a named event. In order to detect collaborative events however, Mingo contains rule composition operators similar to the ones found in rule-based languages. A Mingo rule consists of a name, the left-hand side (LHS) which contains *conditions* for the detection of a complex event, a $\rightarrow$, and a right-hand side (RHS) for *reactions* after the complex event is detected. We introduce the rule for the collaborative resize from our running example in Listing 3. The rule is defined in the application in an object literal using Mingo's `rule` construct which packages the rule. The construct which exposes syntax that developers to write custom complex interactions declaratively.

Listing 3. A Collaborative Resize Rule in Mingo

```
1  rule('collabResizeRule',
2    ' (Invoked (function "touchMove") (dev ?d1) (args ?a1) (time ?time1))
3     (Invoked (function "touchMove") (dev ?d2) (args ?a2) (time ?time2))
4     (test (time:within ?time1 ?time2 1000))
5     →
6     (assert (collabResize (args ?a1 ?a2) (dev ?d1 ?d2))')
```

The `rule` construct expects two arguments: the name of the rule (which denotes the name of the collaborative interaction) and the rule definition – a set of LHS statements followed by a set of RHS statements.

The LHS of the definition (lines 2-4) captures the invocation of simple events from the different entities of the mobile application. This rule captures two `touchMove` invocations and their arguments from two different devices (lines 2-3) that occur within a certain amount of time (line 4).

When all the conditions specified in the LHS are satisfied, then the actions defined in the RHS are activated. Here, we store a new fact `collabResize`(line 6), which is the composition of the two invocations and their arguments. With this assertion Mingo triggers a collaborative event and propagates it to the devices as a collaborative action. In the LHS and RHS of a rule, the `?` operator denotes a variable binding (e.g. `?dev1` in line 2).

The collaborative application can react to a collaborative action by registering a listener. Recall from the previous section that Mingo creates a group object to replicate actions on a shape (thereby updating its changes) to all participants. On this group object we register a listener to the group object by using the Mingo `on` construct, as in Listing 4. The first argument to the `on` construct is a string denoting the collaborative interaction, and the second is a callback that will be invoked once the interaction is achieved.

Listing 4. Reacting to a Collaborative Resize

```
1  groupShape.on('collabResize', function(args){
2    //perform resizing of shape
3    collaborativeResize(this, args);
4  });
```

The shape and its replicas can now receive coordination support. If there is *only* one client device interacting then a normal, single action (e.g. a move) will be invoked. When Mingo detects two actions (e.g. 2 touch downs) from different devices, it will treat this as a collaborative interaction provided it meets the conditions in the LHS part of the respective rules of the shape. On a shared tabletop, if the device can distinguish inputs from different hands [4] we can create a rule condition take this into account: for instance a test that checks the `handIds` are different.

Managing States in Mingo

Composing complex events such as collaborative interactions is simple when done as single-state events e.g. detecting when two participants both perform a touch down interaction. Most collaborative interactions, however, work in a *multi-state* pattern e.g. recognizing a complete gesture done between touchDown and touchUp interactions. We observed that writing single-state rules for a these kinds of higher-level events leads to an inherent complexity in their rule definitions and additional orchestration logic is needed in the application code. This is especially because when composing events into complex events using rules, some inaccuracies that cause arbitrary behaviour when a series of events are detected might delude the developer.

Mingo solves this problem by providing a structured way to define such *multi-state rules* to detect a series of complex events in addition to single ones. Each collaborative operation in Mingo can be represented as a *state machine* that will compose invocation of multi-state events. A *state* in this case is a phase in the detection of a complete composite event. Each of these phases can be of interest to a developer to react to in an application. Mingo rules provide the necessary support to compose the state machine relating to such a composite event. In order to exemplify the problem, consider again the collaborative resize from Listing 3. Whenever a collaborative drag is performed on a shape, a developer has to first check if a collaborative touch down has been performed by the two of the participants before proceeding to listen for a collaborative resize. To implement the detection of the interaction in stages, we redefine the collaborative resize interaction in terms of the following states:

1. **Resize start** – When the two participants perform a `touchdown` at almost the same time; the collaborative interaction has begun.
2. **Resize body** – After resize start, when one or both participants start (and continue) performing a `touchmove` in opposite directions.
3. **End** – When one or both participants perform a `touchup`, the collaborative interaction comes to an end.

The states can be encoded in Mingo as shown in Listing 5. The `collabResize` composite event now consists of the three aforementioned states (denoted in the code as `start`, `body` and `end` states). `collabResize` is actually an object that simply contains a start, body and an end state defined as Mingo rules. A developer can thus detect any number of multi state operations separately – such as the *start*, *body* and *end* of the collaborative resize operation – and execute code in their respective handlers.

Listing 5. Collaborative Resize in Mingo - With States

```
1  collabResize = {
2     start: rule('collabResizeStart', /* rule definition here */),
3     body: rule('...'), /* rule indicating the body */
4     end: rule('...') /* rule indicating the end */
5  };
6  groupShape.on(collabResize.start, function(args){
7     //handle resize start operation
8  });
```

Note that the developer can react to any of these states of the collaborative resize separately with a handler on each one. For example, lines 7-9 show the reaction to a collaborative resize event for the `collabResize.start` rule on the shape object of our running example.

In the next section we illustrate how a developer defines a multi-state rule.

Defining a Multi-State Collaborative Rule

Mingo provides an extended version of the `rule` construct which allows developers to define a rule that is only triggered when the application detects a concrete state of an extended interaction. Listing 6 shows such a rule for the `start` state of the collaborative resize seen in the previous listing Listing 5.

Listing 6. Structure of a Multi-state Rule

```
1  //SP
2  ?currstate ← (State (name "end") (operation "collabresize"))
3  //IP
4  (Invoked (function "touchDown") (dev ?d1) (args ?a1) (time ?t1))
5  (Invoked (function "touchDown") (dev ?d2) (args ?a2) (time ?t2))
6  (test (time:within ?t1 ?t2 1000))
7  →
8  //SS
9  (retract ?currstate)
10 (assert (State (name "start") (operation "collabresize") (args ?a1 ?a2)))
11 //CB
12 (assert (collabResizeStart (args ?a1 ?a2)))
```

The `collabResizeStart` rule has the LHS and RHS as defined earlier. Every multi-state rule is further defined in a four-part structure, each of which have specific purposes.

On the left-hand side (LHS) of the multistate rule we have:

State predicates (SP) – The conditions about the state of the operation needed to detect the composed event. This is the first part where we can detect any states that are needed prior to the execution of this rule. For instance, in line 2 we avoid multiple activation of a state by making sure that any previous state must currently be in the `end` (or idle) state before transitioning to the `start` state. Mingo provides the '←' symbol binds the `end` state to the `?currstate` variable, which can be used to modify the current state within the rule, as shown in the third part.

Invocation predicates (IP) – Conditions about the invocations required to compose the event. This describes the invocations that need to be detected in order to realize the composed event. In lines 4-5 we detect that two touch down operations are performed as the invocation conditions for the `collabResize` start operation, which we defined in Listing 5.

And on the right hand side (RHS) we have:

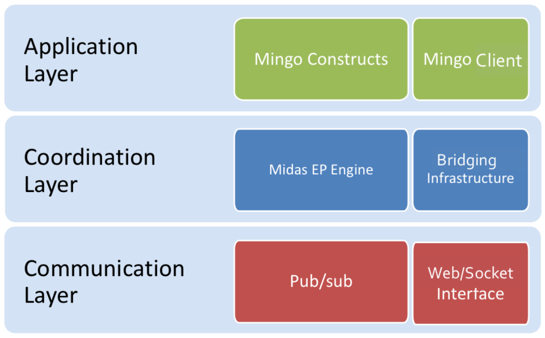


Figure 4. Mingo's layered architecture

State switching (SS) – Change in states of the operations (e.g. `start`, `body` and `end`) based on the current state. Here we provide the logic that will transform the state of the composed complex event represented in this rule from one state to another. This is often done simply by the *retraction* or deletion of the current state (in line 9, using the bound variable `?currstate` from line 2 representing the `end` state); and the addition or *assertion* of a new state (`collabresize` start state in line 10).

Callback (CB) - The callback part consists of one or more *asserts* that produce the collaborative event. Here we can deliver all the arguments that the callback may need for handling the event. In this case, Mingo adds the `collabResizeStart` fact, and triggers the associated callback of this event in the application (line 12). Note that this is different from the change of state, since this assert actually triggers the collaborative interaction to all clients.

It is important to note that the multistate rules allow developers to coordinate collaborative multistate interactions while avoiding the complexity introduced by composing rules with multiple states. Without support for multistate rules, the developer will be forced to manually maintain state logic within the handler whenever a collaborative event is realized.

IMPLEMENTATION

We now have a look at the general Mingo architecture. The structure is realized as a layered architecture, depicted in Figure 4. This is largely due to the target implementation technology that is the web, which imposes a layered architectural style. We briefly discuss each of the layers of the stack in detail in the rest of this section.

Communication Layer

The Communication Layer in Mingo is tasked with handling the transportation of any possible event information from the Application Layer to the Coordination Layer. For the introductory example, this layer would handle the multicasting of all drawing interaction information to the event processing engine, as well as to other participants in the session. Mingo utilizes techniques from the publish/subscribe paradigm in order to broadcast this event information to the rest of the participants and to the Midas engine. This multicasting is propagated using a Node.js server, through the use of HTML5 websockets. When an object is tagged as a `groupObject` in Mingo, the actions that relate to a collaborative interaction on the object will be replicated to the rest of mobile devices in the session.

Coordination Layer

The coordination layer handles the actual internal distributed coordination mechanism of Mingo. It contains the event processing engine Midas, the Node.js server and the bridging infrastructure needed for internal event subscription and detection. The Mingo runtime employs an extension of Midas to provide this kind of bridging interface to register and listen to activations of complex events as well as to replicate actions on object groups, thus utilizing the engine for coordinating collaborative interactions.

To coordinate collaborative events, every action on a `groupObject` that may contribute to a collaborative interaction is sent to the server as an event. Mingo transparently translates all simple interactions or events are into temporal facts in the Midas engine. When a fact is asserted it can trigger a collaborative event if all the conditions of the rule hold. In this case Mingo will then invoke the respective collaborative interaction in all the mobile clients in the session – through the callback provided as the second argument of the `on` construct.

Application Layer

For the previous two layers to be helpful to the developer for creating a collaborative application, there needs to be a suitable way to expose their functionality. The Application Layer in Mingo provides the necessary abstractions for developers of web-based mobile applications that make it simple to introduce collaborative mechanisms to their applications using the `on` and `groupObject` constructs. The Mingo client intercepts calls to operations on objects and sends them to be stored in the CEP engine.

When an object is tagged as a `groupObject`, Mingo assigns a unique identifier to the object and proceeds to create replicas in the other mobile devices connected to the session. Additionally, Mingo detects the rule defined and sends it to the server to be stored in the event processing engine Midas. The engine stores pre-defined rules with the aim of detecting a possible distributed collaborative event in the mobile application through the actions of the participants in the current session. Subsequently, using the `on` construct on such a replicated object will cause the callback assigned to be invoked whenever the engine triggers a collaborative event as defined by its respective rule. The application layer thus exposes these mechanisms to the developer of a collaborative mobile application.

EVALUATION

In this section we demonstrate how Mingo's language support eases the development of complex cloud-based applications

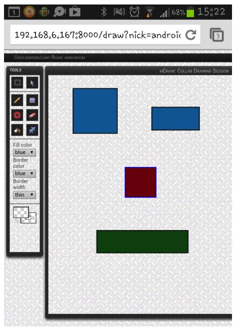


Figure 5. mDraw mobile application

by implementing a collaborative drawing mobile application. We then implement a similar application supported by a rule based approach for the web. We subsequently compare the two approaches and their support for implementing collaborative interactions.

i) Collaborative Drawing Application: mDraw

mDraw is the concrete implementation of the collaborative drawing application introduced in the motivating example. It runs on mobile browsers supporting JavaScript and HTML5. To join a session, clients simply connect to the application using a url. The application code of the web app can be cloned from Github[7].

Having explained the collaborative resize operation used in mDraw, we now explain the implementation of the *collaborative bind* operation from the second motivating example of collaborative interactions and illustrated in Figure 1 (b). This interaction consists of a `start` state when two participants touch the canvas, a `body` state where they continue dragging in opposite directions and `end` state where the canvas borders have been touched by both participants. Since the start and body states are similar to the collaborative resize interaction, we illustrate the implementation of the completed `end` state in Listing 7.

Listing 7. mDraw: Collab Bind Start Rule

```
1  rule('bindEndRule',
2      '?bodystate ← (State (name "body") (operation "bind") )
3      (Invoked (function "touchMove") (args ?x1 ?y1) (time ?t1)))
4      (Invoked (function "touchMove") (args ?x2 ?y2) (time ?t2)))
5      (Canvasbounds (right ?right))
6      (test (and (>= ?x1 ?right) (<= ?x2 0)))
7      (test (and (<= (abs (− ?y1 ?y2)) 10) (time:within ?t1 ?t2 1000)
8      →
9      (retract ?bodystate)
10     (assert (State (name "end") (operation "bind")))
11     (assert (bindEnd (args ?x1 ?y1 ?x2 ?y2)))')
```

To detect when the bind action has been completed we need to first check if a bind was already detected (line 2) and that it is currently in the body state. Next we check if there are two `touchMove` operations (line 3-4) that have reached the opposite sides of the border of the canvases in the two devices (lines 6-7). We change the state to a bind end state (line 9-10), and finally assert that a completed collaborative bind interaction has been realized (line 11). The application can then react to the assertion of this event by invoking its callback.

[7] `http://bit.ly/mingomdraw`

ii) Implementation in JavaScript using Nools

In this section we employ mDraw to evaluate our approach by comparing it with a similar ad-hoc implementation written in JavaScript.

We identified Nools, a JavaScript rule engine as the closest to our declarative approach for the cloud. Nools enables developers to specify complex interactions as logic rules in their applications and trigger events though callbacks. Nools can run in the server as a Node.js package or in the desktop client as a library.

In order to provide functionality for collaborative interactions using Nools, we were forced to provide static rule definitions on the server. In Listing 8 we show the relevant sections when defining collaborative bind interaction programmatically in Nools.

Listing 8. Nools: Reacting to a Collaborative Bind

```
1  var collabBind = flow.rule("bindEnd", [
2      [Invoked, "inv1", "inv1.function == 'touchMove'", {x: "x1", y: "y1", time:
              "t1"}],
3      [Invoked, "inv2", "inv2.function == 'touchMove' && (Math.abs(y1 − y2)
              <= 10) && (Math.abs(t1 − t2) <= 1000)", { x: "x2", y: "y2", time
              : "t2"}],
4      [Canvasbounds, "b", "(b.right <= x1) && (x2 <= b.left)", {right: "?right",
              left: "left"}],
5      [State, "state", "state.name == 'body' && state.operation == 'bind'"]
6      ],
7      function(facts){
8        var inv1 = facts.inv1, inv2 = facts.inv2;
9        var args = [facts.args1, facts.args2];
10       this.assert(new State('end', 'bind', inv2.time));
11       this.assert(new bindEnd(args, inv2.time);
12       clientManager.sendCollabEvent('collabBind', [inv1.dev1, inv2.dev2],
               args);
13     }
14   );
```

Because nools is a rule-based language, the rule structure is similar to that of Mingo rules. Line 1 defines the collaborative bind detection operation using Nools' `flow.rule` construct. The construct takes an array that contains rule definitions as the first argument. In lines 2,3 we detect if the touchMove operation has occurred from the client browsers within a time range of 1000 milliseconds. Line 4 detects if the operations have both touched the canvas borders and line 5 is an additional check to see if the bind operation already begun in its previous body state.

The third argument to `flow.rule` is a callback (line 7), which is invoked whenever the rule fires. The callback is invoked with the facts that triggered the rule as arguments. We use the arguments to assert the new state of the collaborative interaction (line 10-11), similar to the states in the multistate rules. We then call the code that will eventually send to the participants that the collaborative interaction has been detected, sending all the arguments in an array (line 12).

Discussion

We proceed to compare the approaches provided by the Mingo framework and the Nools engine. We begin by a lines of code (LOC) quantitative comparison illustrated in

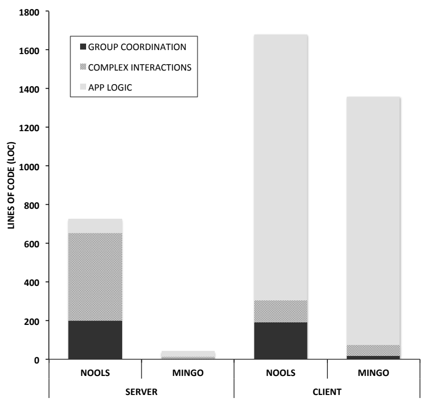


Figure 6. LOC Comparison of Sample Application

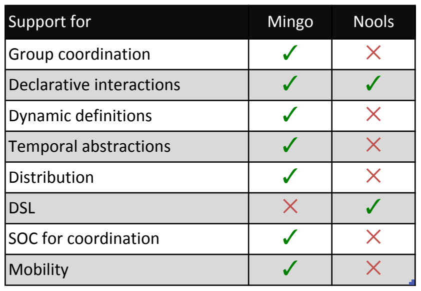
Figure 6. We measured the code on the server and client sides in terms of group coordination (replication, communication, session management), complex interactions (collaborative events definitions, detection and activations) aside from the rest of the application logic. We find that the Mingo framework reduces the code a developer writes to enforce collaborative interactions – particularly on the server side where the orchestration of events is implemented.

Next we qualitatively compare each of the two approaches in terms of support of various coordination features (illustrated in Table 7). For *group coordination*, Mingo maintains consistency by transparently replicating specified actions to all connected mobile clients within the same session while Nools provides no mechanism for this. Both approaches are rule-based, they therefore support definition of complex interactions *declaratively*.

In Nools, the rules need to be statically defined in order to be utilized in an application. This limits the *dynamism* of rule definitions. Dynamically adding rules enables developers to add separate rules for different groups of replicated objects thereby providing specialized types of interaction on different shapes (e.g. we can dynamically define separate collaborative interactions for rectangles and circles). Through Midas, Mingo natively supports temporal abstractions such as the `time:within` construct while in the Nools approach we had to specify time calculations and assert the facts with temporal annotations.

Mingo rules can be defined by a developer on the mobile client side and registered to be processed on the server by the runtime. Nools natively has no support for this *distribution*; when one defines rules on the browser they work only in the browser, and if on the server then only the server code can utilize the rules.

In our sample application for Nools, we observed that most coordination code was interspersed with the application logic for the drawing editor and the server code, making debugging difficult and modification prone to unexpected errors. This goes against the principle of *separation of coordination concerns*. Mingo code isolates the coordination logic, making development less vulnerable to such limitations.



Support for	Mingo	Nools
Group coordination	✓	✗
Declarative interactions	✓	✓
Dynamic definitions	✓	✗
Temporal abstractions	✓	✗
Distribution	✓	✗
DSL	✗	✓
SOC for coordination	✓	✗
Mobility	✓	✗

Figure 7. Feature Support Comparison

Nools does provide a *DSL* (Domain-Specific Language) for 'cleaner' rule definition syntax, while Mingo does not. However, the Nools rules defined with the DSL need to be compiled before beforehand. A final point is that Nools lacks support for *mobile* devices.

LIMITATIONS & FUTURE WORK

Mingo utilizes HTML5 websockets for communication, identified as one of the best performing web technologies for collaborative groupware [8]. Furthermore the performance of the Midas CEP engine in processing real-time event streams has been previously documented in [14]. In the prototypes we developed and tested the applications were quite responsive. Nevertheless, Mingo is currently restricted to low-latency environments since it is constrained to the inherent limitations of the distributed web architecture. For instance, when several simultaneous interactions from different clients are asserted to the server, their effects may cause the replicas to diverge. In future we intend to investigate how enforcing techniques such as eventual consistency with operational transformation [18] will ensure that the operations *eventually* converge on all the clients, thereby relaxing this restriction.

We would also like to investigate how Mingo can estimate and handle small delays in communication between clients and server with respect to the defined rules, which affects detection of delayed events. Finally, we intend to create a DSL for more intuitive definitions of Mingo's collaborative rules both on the server and in the mobile clients.

CONCLUSION

We have focused on programming support for interactive web-based mobile applications in which a number of clients participate in performing collaborative surface interactions. We identified requirements that a suitable programming model for coordinating these kinds of interactions should adhere to: event composition and group coordination.

We propose Mingo, an object-oriented framework that extends JavaScript with coordination abstractions that blend complex event processing and group coordination. Mingo allows developers to cohesively compose events coming from distributed client devices by defining rules and to coordinate behaviour of distributed objects by defining group constructs. Furthermore, the framework enables developers to program their coordination concerns separate from their application logic. The major contribution of Mingo is to merge techniques found in declarative complex event processing and distributed group coordination into a coherent object-oriented framework for collaborating interactions in mobile applications running on web architecture.

REFERENCES

1. Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, and Mitch Cherniack. 2005. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research. (CIDR'05)*. Asilomar, CA.
2. Sriram Karthik Badam and Niklas Elmqvist. 2014. PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14)*. ACM, New York, NY, USA, 109–118. DOI: `http://dx.doi.org/10.1145/2669485.2669518`
3. Jacob T. Biehl and Brian P. Bailey. 2004. ARIS: An Interface for Application Relocation in an Interactive Space. In *Proceedings of Graphics Interface 2004 (GI '04)*. Canadian Human-Computer Communications Society, 107–116. `http://dl.acm.org/citation.cfm?id=1006058.1006072`
4. Florian Echtler, Manuel Huber, and Gudrun Klinker. 2008. Shadow Tracking on Multi-touch Tables. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '08)*. ACM, New York, NY, USA, 388–391. DOI: `http://dx.doi.org/10.1145/1385569.1385640`
5. Charles Forgy. 1982. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligences* 19, 1 (1982), 17–37. `http://dx.doi.org/10.1016/0004-3702(82)90020-0`
6. D. Gelernter. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 7, 1 (Jan 1985), 80–112.
7. Yechezkal-Shimon Gutfreund and John R Nicol. 1997. WWWinda Orchestrator: a mechanism for coordinating distributed flocks of Java Applets. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series (Electronic Imaging'97)*, M. Freeman, P. Jardetzky, and H. M Vin (Eds.). 295–302.
8. Carl A. Gutwin, Michael Lippold, and T. C. Nicholas Graham. 2011. Real-time Groupware in the Browser: Testing the Performance of Web-based Networking. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work (CSCW '11)*. ACM, New York, NY, USA, 167–176. DOI: `http://dx.doi.org/10.1145/1958824.1958850`
9. Ken Hinckley. 2003. Synchronous Gestures for Multiple Persons and Computers. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. ACM, New York, NY, USA, 149–158. DOI: `http://dx.doi.org/10.1145/964696.964713`
10. Brad Johanson, Greg Hutchins, and Terry Winograd. 2000. *PointRight: A System for Pointer/Keyboard Redirection among Multiple Displays and Machines*. Technical Report.
11. Jonathan P. Munson and Prasun Dewan. 1997. Sync: A Java Framework for Mobile Collaborative Applications. *Computer* 30, 6 (June 1997), 59–66. DOI: `http://dx.doi.org/10.1109/2.587549`
12. George A. Papadopoulos and Farhad Arbab. 1998. Coordination Models and Languages. In *Advances in Computers. (The Engineering of Large Systems)*. Academic Press, New York, NY, USA, 329–400.
13. Mark Proctor. 2012. Drools: A Rule Engine for Complex Event Processing. In *Applications of Graph Transformations with Industrial Relevance*, Andy Schürr, Daniel Varrò, and Gergely Varrò (Eds.). Lecture Notes in Computer Science, Vol. 7233. Springer Berlin Heidelberg. DOI: `http://dx.doi.org/10.1007/978-3-642-34176-2_2`
14. Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. 2012. Parallel Gesture Recognition with Soft Real-time Guarantees. In *Proceedings of the 2Nd Edition on Programming Systems, Languages and Applications Based on Actors, Agents, and Decentralized Control Abstractions (AGERE! 2012)*. ACM, New York, NY, USA, 35–46. DOI: `http://dx.doi.org/10.1145/2414639.2414646`
15. Mark Roseman and Saul Greenberg. 1992. GROUPKIT: A Groupware Toolkit for Building Real-time Conferencing Applications. In *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work (CSCW '92)*. ACM, New York, NY, USA, 43–50. DOI: `http://dx.doi.org/10.1145/143457.143460`
16. Oliver Schmid, Agnes Lisowska Masson, and Béat Hirsbrunner. 2012. Collaborative Web Browsing: Multiple Users, Multiple Pages, Concurrent Access, One Display. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12)*. ACM, New York, NY, USA, 141–150. DOI: `http://dx.doi.org/10.1145/2305484.2305508`
17. C. Scholliers, L. Hoste, B. Signer, and W. De Meuter. 2011. Midas: A Declarative Multi-Touch Interaction Framework. In *Proceedings of 5th International Conference on Tangible, Embedded, and Embodied Interaction. (TEI'11)*. Funchal, Portugal.
18. Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW '98)*. ACM, New York, NY, USA, 59–68. DOI: `http://dx.doi.org/10.1145/289444.289469`