A Tuplespace Event Model for Mashups

Sheng Tian
School of Engineering
University of Auckland
New Zealand
shengt@gmail.com

Gerald Weber, Christof Lutteroth
Department of Computer Science
University of Auckland
New Zealand
{gerald, christof}@cs.auckland.ac.nz

ABSTRACT
Inter-widget communication is essential for enterprise mashup applications. To implement it, current mashup platforms use the publish/subscribe pattern. However, the way publish/subscribe is used in these platforms requires a lot of manual wiring between widgets. In this paper, we propose a new Unified Widget Event Model (UWEM), which is conceptually an extension of Linda tuplespaces. UWEM separates event publishers and subscribers in space, time, and reference. Using the Keyboard-Level Model (KLM) we show that UWEM requires fewer operations to build typical mashups than conventional mashup platforms. We have implemented UWEM in a popular enterprise mashup framework, and performed an empirical study that compares UWEM with the established approach for creating mashups. The study confirms the KLM predictions, and shows that UWEM is significantly more efficient than the established approach.

Author Keywords
Tuplespace, mashup, widget, event model

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
One of the most empowering user interface technologies currently available on the Web is that for creating mashups. A mashup is a web page or application that uses and combines data, presentation or functionality from two or more sources. Mashups integrate content and functionality that is available through open Web APIs (application programming interfaces) and reusable services. They were initially conceived as a means for business users to create their own applications, starting from public APIs such as Google Maps. Today, a large amount of content, such as photos, videos, news, maps, weather forecasting, and e-commerce can be combined in a mashup.

A mashup application is a situational application, i.e. "good enough" software created for a narrow group of users with a unique set of needs. Unlike traditional applications, situational applications try to solve those requirements on the long tail of all the enterprise requirements. These applications are usually too costly to build using the traditional way, or have a lower priority than the strategic applications of an enterprise, as shown in Figure 1.

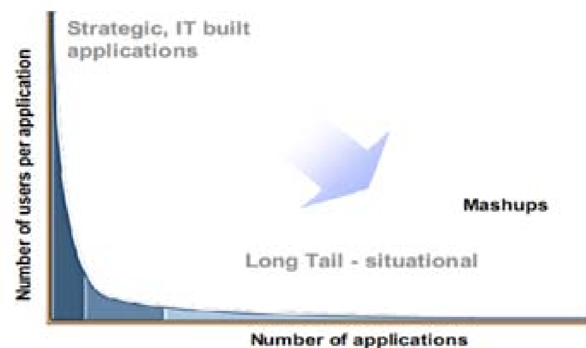

Figure 1. Situational applications (Carrier, 2008)

Mashup building is a kind of integration and composition technology. Integration itself has been a main focus of software development methods and technologies for the last 30 years. We can distinguish between two main classes of mashups: *data mashups*, and *presentation mashups*. These classes correspond to two important classes of integration: data integration and user interface (or, more broadly, presentation) integration. In this paper, we focus on the *presentation mashups*.

Presentation mashups use widgets as UI components. A widget is an interactive single-purpose application that can be installed and executed by an end user. By wiring multiple widgets, data and applications can be integrated quickly and easily, effectively creating new applications that provide an added value in their own right.

Widgets run in widget containers (Sire, 2009), which are the widgets' runtime environments. In order to support interaction and communication between widgets, models for events and event handling have been introduced. Inter-widget communication (IWC) is essential in *presentation mashup* applications, as it allows widgets to communicate with and react to other widgets. However, the concepts of widget events and event handling are often considered too complex for end users, and therefore they are often only made available to developers. The aim of our research is to make IWC also accessible for the average end user.

We will use the following running example in this paper. Pat is going to Auckland to attend a conference next week.

OZCHI 2011, Nov 28 – Dec 2, 2010, Canberra, Australia.
Copyright the author(s) and CHISIG
Additional copies are available at the ACM Digital Library (http://portal.acm.org/dl.cfm) or ordered from the CHISIG secretary (secretary@chisig.org)
OZCHI 2011 Proceedings ISBN: 978-1-4503-1090-1

She needs to know the weather in Auckland, and wants to find a hotel that is close to the conference. She builds a mashup application to help her plan the travel, which is shown in Figure 2. First, Pat puts a *conference detail widget* (A) on the page to show the conference. Then she adds a *hotel search widget* (B), and wires it with the *conference detail widget*. As a result, the *hotel search widget* accepts the conference address from the *conference detail widget* as its input parameter and shows all hotels nearby on the map. Finally, Pat adds a *weather forecast widget* (C) that accepts the conference address and date as its input parameters, showing the weather forecast during the conference in Auckland.

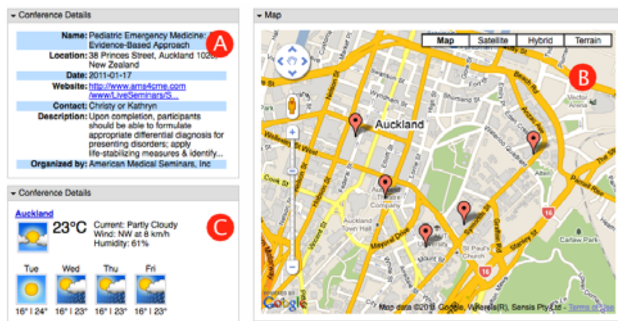

Figure 2. Conference mashup scenario

In this example, the conference location information is broadcasted. *Widget B* receives the location and shows hotels nearby on the map. *Widget C* receives the location and shows information about the weather in the area. For the sake of illustration, the conference date is unicasted to *widget C*, as *widget C* needs both the location and the date from *widget A* to query data from a weather forecast web service. When changing the conference details, the *hotel search widget* and the *weather forecast widget* will be changed accordingly. Hence, the above mashup application can also be used to plan other conference travels, so Pat can share it with her colleagues.

Current widget specifications have many limitations when it comes to IWC. Many widget specifications only support unicast interaction, e.g. *IBM Mashup Center iWidget* (Ketter et al., 2009). Other specifications, such as iGoogle Gadgets (Casquero et al., 2008) and Netvibes UWA (Kaar, 2007) require additional middleware to support IWC. All IWC implementations are unique and incompatible with each other.

We argue that creating the mashup in the given example with current IWC technologies is not as easy as it should be. The wiring that was manually created in the example above is the natural default for that scenario; hence there should be a larger degree of automation in creating it. We propose a new and extended event model, the Unified Widget Event Model (UWEM), which achieves a much higher degree of automation than existing approaches. UWEM satisfies the following requirements:

- **Flexible Communication Patterns**: An event model should support different routing schemes, including unicast, multicast and broadcast, so that it can be used in the existing mashup platforms.
- **Automatic Configuration**: An event model should enable a "plug and play" approach to support automatic wiring between the widgets in a mashup, in a way that makes manual wiring unnecessary for most applications.

The new widget event model is based on Linda tuplespaces (Gelernter, 1985). Linda is a programming language that is intended for distributed systems. Messages are added in the form of tuples to a global data store called a tuplespace (TS). In the TS, tuples exist as named independent entities that can be read and consumed by processes. Linda is fully distributed in both time and space, and by introducing TS into mashup applications, the communication between widgets becomes decoupled in time, space and reference. The TS approach to events in mashups is compatible with the existing IWC models, so that existing applications can be integrated with the TS. Furthermore, the TS approach makes it possible to build mashups with minimal or zero manual wiring.

The paper is organized as follows. The next section presents the capabilities and limitations of the communication models used in the current mashup frameworks. Afterwards, we introduce the Unified Widget Event Model (UWEM), and describe how the communication patterns of existing frameworks can be specified in the new model. Then, we present an evaluation that compares UWEM with existing event models, and we discuss the advantages and disadvantages of the new model. Finally, we offer conclusions and point out future directions.

RELATED WORK

There are many areas of research involving IWC event models. The following sections give an overview of the history of event models, different routing schemes, communication patterns, and the main approaches of IWC. This overview helps to understand why IWC is often considered too hard to understand for end users. It also shows what functionality a new event model such as UWEM needs to consider in order to be useful for real applications.

Events and Event Models

The concept of an event is used in different ways in different software contexts. For example, GUI operating systems use events to interact with applications. In concurrent programming, programs set and reset events, which are process synchronization primitives such as semaphores and mutual exclusions (mutexes); events are basically flags that one process can set and another can read. A popular object-oriented design pattern, the observer pattern, is based on events. Programming languages (e.g. .Net, Java) also use the term event to denote actions or entities of various kinds, in which events are often associated with event sources, event objects, and event listeners.

Publish/subscribe based event models were first introduced in the data and business domain as complex event processing (Rosenblum and Wolf, 1997; Cugola

and Margara, 2011). These models support content-based filtering mechanisms. The programming language community investigated language primitives for supporting publish/subscribe models (Eugster, 2001). In this approach, events are treated as first-class objects in an object-oriented programming language, so they have a type and can be processed like ordinary data. Subscribers specify the class of objects they want to receive.

In general, an event is defined as an instantaneous, atomic (i.e. happening completely or not at all) occurrence of interest (Chakravarthy, 1994). An event model is a software architecture that determines how components can: create and describe events, trigger events, distribute events to interested components, subscribe to event sources, react to events when received, and remove the subscription to event sources when desired.

Cugola identified three classes of event-based systems according to event structures (Cugola, 2002):

- *Events as Tuples*, where fields are distinguished by position. Linda TS based communication falls into this category.
- *Events as Records*, i.e. sets of typed fields characterized by a name and a value. Within this category, different event-based infrastructures can be further classified depending on the richness of the type system they offer. These event systems are seen mostly in Web application, in which components use JSON as a record-based event structure (Crockford, 2006).
- *Events as Objects* that have both a state and a set of methods.

The structure of events has an impact on how easily they can be understood by users, as opposed to developers. For example, events as objects may require a user to understand programming concepts such as methods. Tuples may be familiar to users as a concept from elementary mathematics.

Event Dispatcher Architectures

An important factor that has an impact on performance and scalability of event-based infrastructures is the internal architecture of the event dispatcher. Cugola classified event dispatcher architectures according to the following categories (Cugola, 2002):

- *Direct Connection*. No explicit event dispatcher exists. Events are directly dispatched by the event sources to the interested parties. This architecture is popular in unicast environments.
- *Broadcast*. Events are always sent out to all parties.
- *Centralized*. There is a single dispatcher that all events are sent to, which takes care of the deliveries to the interested parties.
- *Distributed*. A number of interconnected dispatching servers cooperate to deliver events.
- *Mixed*. This refers to a combined approach, such as the use of broadcasting to deliver events within certain groups of nodes, and the use of other approaches to deliver them between the groups. Such a mixed approach is used, for example, in networks: messages within a LAN are broadcast, whereas a different approach is used to deliver messages between different LANs in a WAN.

While the choice of event dispatcher architecture seems to be a very technical decision, it does have consequences for mashup developers. More complex architectures make it harder to understand - and change - the flow of events in an application.

Routing Schemes

The notion of routing as known from networks (Medhi, 2007) can also be used to characterize IWC. Routing schemes differ in their delivery semantics:

- *Unicast* delivers a message to a single specified node.
- *Multicast* delivers a message to a group of nodes that have expressed interest in receiving the message (Ramalho, 2000). The terms multicast and narrowcast are often used interchangeably, although narrowcast usually refers to the business model whereas multicast refers to the actual technology used to transmit the data.
- *Broadcast* delivers a message to all nodes in the network.
- *Anycast* delivers a message to any one out of a group of nodes, typically the one nearest to the source (Abley, 2006).

Many event models implement unicast, multicast and broadcast. As a consequence, a new event model such as UWEM should support at least these three routing schemes.

Inter-Widget Communication

The idea of inter-widget communication (IWC) is not new. Sire proposed a JavaScript messaging API for inter-widgets communication (Sire, 2009) that wires widgets with a drag-and-drop metaphor within the browser. Wu proposed a similar web widget communication framework (Wu, 2010). In both approaches events can be predefined browser events, or application-specific events that are defined by the developer. Event data are passed between widgets when events associated with those widgets occur.

Several professional mashup solutions with IWC features were proposed. Google provides a gadget-to-gadget communication framework in its iGoogle Gadget specification (Casquero et al, 2008). The publish/subscribe framework allows publisher gadgets on iGoogle to communicate changes to subscriber gadgets that have declared interest in those changes.

IBM Mashup Center provides a similar publish/subscribe framework called iEvent. For every iEvent, the widget container has to explicitly declare a wiring relationship between the source widget and the target widget. iEvent is compatible with OpenAjax Hub (OpenAjax Alliance,

2009), which is a client-side Ajax component communication standard based on publish/subscribe.

The IWC provided by iGoogle Gadgets and IBM iWidgets are limited. Both support only event distribution patterns that can be modeled with the publish/subscribe mechanism. Furthermore, widgets communicating on these platforms are all tightly coupled in time, which means messages cannot be delivered if the receiver widget has not yet been put into the mashup application.

BISSA (Wickramasinghe, 2010) is an IWC solution specifically for Google gadgets that is related to UWEM. It introduces a TS into the browser, and allows gadgets to coordinate themselves with other gadgets through the TS. However, BISSA does not address our requirements. Compared to UWEM, BISSA addresses implementation issues of a TS on a lower level. Most importantly, it does not answer the questions of how to simplify wiring for end users. In principle, UWEM could be implemented on top of BISSA - something that we might investigate in the future.

UNIFIED WIDGET EVENT MODEL

The Unified Widget Event Model (UWEM) extends the Linda TS with new features to make it suitable for mashup environments. Figure 3 shows a system overview of UWEM. It includes two parts: the first part is the extended TS (blue area in Figure 3), which is implemented with the Dojo toolkit as a platform-independent JavaScript library. This library can be used in any web mashup framework. The second part is the integration of UWEM with existing widget specifications (green area in Figure 3). This integration is platform related, and there are implementations for widget specifications of different mashup platforms.

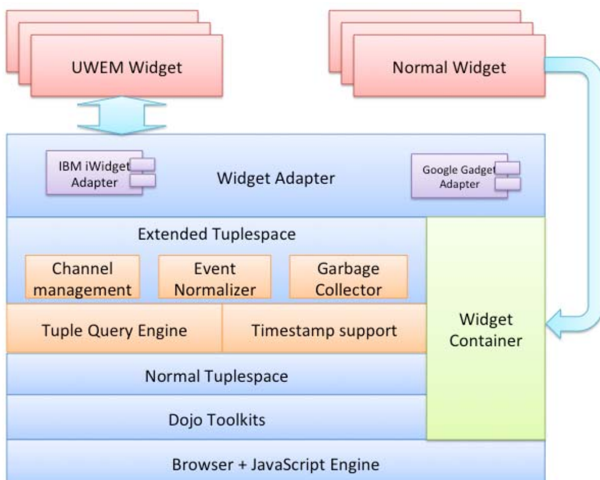

Figure 3. UWEM system overview

In the following, we first describe the original TS concept, and then how UWEM extends it to support communication patterns and automatic configuration.

The Linda Tuplespace

Linda (Gelernter, 1985) is a distributed programming language that provides a model of coordination and communication among several parallel processes. Processes communicate among each other using shared tuples in a TS. Several operations are defined on the TS:

1. *Insertion*: write(N, P2, ..., Pj) inserts the tuple N, P2, ..., Pj into the TS.
2. *Blocking read and delete*: take(N, P2, ..., Pj). If there is a tuple in the TS whose first component is N, then the tuple is withdrawn from the TS. If no matching tuple is available in the TS, take() blocks until one is available.
3. *Blocking read*: read(N, P2, ..., Pj) is the same as the take() statement, but the tuple remains in the TS.

In the above operations, N is an actual parameter specifying a name, and P2, ..., Pj is a list of parameters each of which may be either an actual or a formal parameter. The TS, as the middleware, decouples three orthogonal dimensions involved in IWC:

- *Reference decoupled.* Widgets communicate with each other by writing tuples to and reading tuples from the TS. They do not need to have knowledge of each other in order to communicate.
- *Time decoupled.* Widget communication can be completely asynchronous. A TS stores event data so that it can be read long after the event occurred.
- *Space decoupled.* Widget can be run in different domains as long as they can access the same TS.

UWEM Event Tuples

In UWEM, an event tuple (ET) is a triple:

<event-tuple> := (<event-id>,
<event-metadata>, <payload>)

<event-id> is a string that is globally unique and identifies an ET. <payload> is the event data produced or consumed by widgets. <event-metadata> is a tuple which contains necessary metadata for UWEM to support different message passing patterns. The metadata includes the following information:

- *Source (optional)* is the sender widget's identity.
- *Target (optional)* is the target widget's identity. It can be one widget's identity or a list of identities.
- *Topic (optional)* is a string that is only used during multicasting. With the topic, UWEM can support the *publish/subscribe* communication pattern, as explained later.
- *Timestamp (optional)* encodes the time when the event occurred. It is used for time-aware communication, such as for the message queue communication pattern. This will be explained later on.

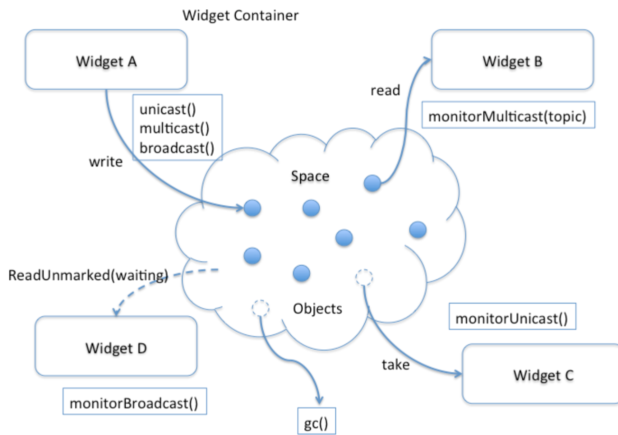

Figure 4. Basic operations of a TS and UWEM

UWEM Operations

UWEM extends the original TS and defines several operations in the widget container to support common event communication patterns. For sending an event, UWEM offers the following operations:

- *unicast(event-tuple)* sends the ET to another widget. The receiver widget is defined in the Target field of the <event-metadata> parameter. After all receiver widgets receive the ET, the ET will be removed from the TS automatically.
- *multicast(event-tuple)* is the same as the publish operation in publish/subscribe event systems. The event topic is defined in the <event-metadata> parameter. The ET will not be removed from the TS until the gc() operation is explicitly invoked.
- *broadcast(event-tuple)* broadcasts the ET to all the widgets in the page, and keeps the ET available in the TS for later widgets. The ET will not be removed from the TS until the gc() operation is explicitly invoked.

The following operations are offered for receiving events:

- *receive(tuple-template)* reads all ETs matching the tuple-template. This is similar to the basic TS operation read().
- *subscribe(topic)* receives an ET that was emitted from a sender widget with multicast(), given that sender and receiver specify the same topic.
- *read/takeLaterThan(tuple-template)* reads or takes an ET from the TS that matches the tuple-template, or matches it except for a later timestamp.
- *read/takeLatest(tuple-template)* reads or takes the latest ET (i.e. with the newest timestamp) that matches the tuple-template from the TS.
- *readUnmarked(tuple-template)* reads an ET from the TS that the caller has not seen before.

Each receiving operation has a blocking version and a non-blocking version. Similar to the original TS operations, a tuple-template can contain actual as well as formal parameters. That is, developers can either specify the value for an ET property to select the ETs they need, or leave it open and read the value when an ET arrives.

Broadcast and multicast ETs will not be removed by any of the operations above, so we need an operation to clean garbage ET periodically:

- *gc(age)* removes all broadcast or multicast ETs from the TS which are older than the given age.

Figure 4 shows the relationship between the UWEM operations and the basic TS operations. All UWEM operations are implemented with the basic TS operations take, read, and write. Each widget has the methods *monitorUnicast()*, *monitorMulticast()* and *monitorBroadcast()*, which can be overwritten to receive ETs that were sent with the corresponding operations.

The operations are implemented in the widget container, and the implementations are therefore platform dependent. However, the operations' method signatures are the same across platforms, so all widgets can use the same interfaces to access them.

Modelling Communication Patterns

The existing publish/subscribe event models support multicast, but do not natively support broadcast and unicast. UWEM supports different routing schemes including *broadcast*, *multicast*, and *unicast*. It can also be extended to support some more advanced features such as *queuing*. This section gives more details of how these patterns are implemented in the TS, so that the requirement for flexible communication patterns can be satisfied.

Unicast transmission is the sending of events to a single widget. In UWEM, the sender widget knows the receiver widget and puts the receiver widget's ID into the ET to indicate that only the widget with given target ID can receive it.

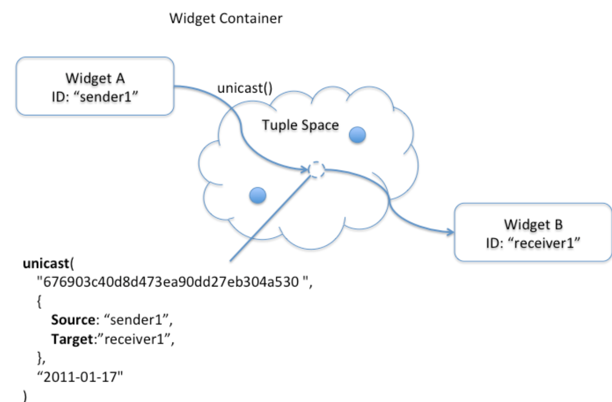

Figure 5.: Unicast in UWEM

Multicast is the delivery of an event to a group of receivers. In UWEM, this can be done using the unicast() and the receive() operations, by providing several target widget IDs as argument for unicast(). The TS collects data about which widget has already read an ET. Once all target widgets have read the ET, the ET is removed from the TS. Alternatively, the multicast() and subscribe() operations can be used, applying the publish/subscribe pattern. Figure 6 illustrates how ETs are published and subscribed by widgets.

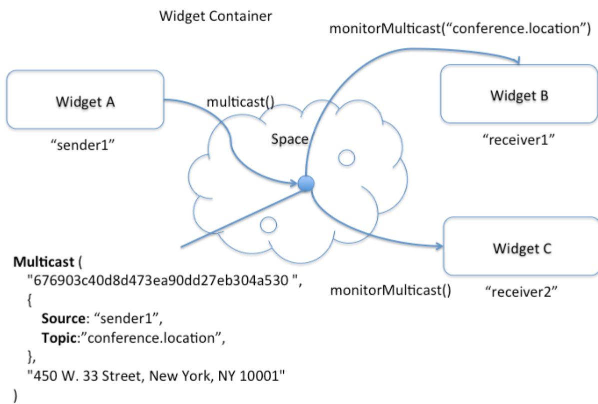

Figure 6. Multicast in UWEM using the publish/subscribe approach

Broadcast is similar to multicast. The difference is that the broadcast ET is transferred to all recipients instead of just some of them. The receiving widgets read the ET without removing it, so that every receiver gets the same copy of the ET. To differentiate broadcast from publish, the Target and Topic fields in the broadcast ET are kept blank, so that all widgets can receive the ET. Old ETs are removed by calling gc().

Queuing. An ET can exist in the TS even after the sender was removed (i.e. time decoupled), as it stays in the TS. Consequently, ETs can accumulate in the TS. For some widgets it is important that the ETs are received in the same order they were sent, e.g. for stock tickers. In the queue pattern, the time order of messages is preserved. UWEM supports queuing by using the Timestamp field of the ETs. The TS makes sure that all read operations return ETs in order of their timestamps. This is illustrated in Figure 7.

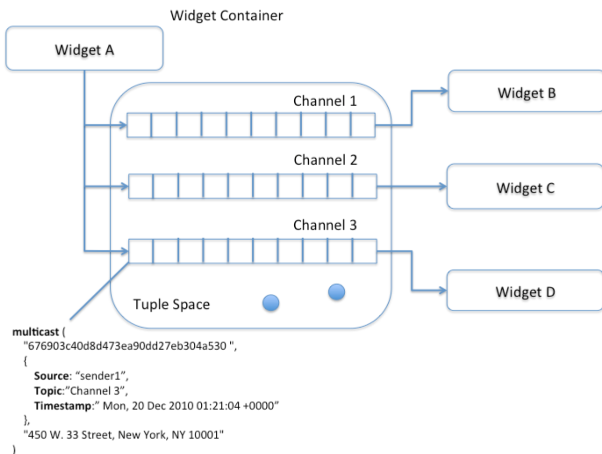

Figure 7. Queuing in UWEM

Automatic Configuration

In current mashup builder implementations, sender and receiver widgets are wired together by widget ID and event topic. Widget wiring needs to be configured by the mashup creator. However, such configuration is not easy to understand and hence error-prone. Even if two widgets are wired together (one subscribes the other's event topic), the communication between them can still fail because the event subscriber does not know if the incoming event can really be consumed. It could happen that an event is received, but does not contain the expected payload. UWEM addresses this issue by using event content sniffing. Unlike the traditional publish/subscribe event model, an ET in the TS is available to receiver widgets before they commit to processing the ET. A receiver can read and test the ET to see if it can be consumed before processing it.

Every widget is preconfigured to listen to a set of topics that it naturally consumes. This means that widgets that produce ETs with a certain topic and widgets that naturally consume ETs of that topic are automatically wired (i.e. "plug and play"). This saves people who create mashups much of the effort and complexity of manual configuration.

Because of the full separation of event sender and receiver, ETs are communicated even if the event source was removed. This makes widgets in mashups more independent from their environment. Adding and removing widgets does not have such a strong impact on the whole mashup application as with conventional event models, where this can be very disruptive. This behavior makes a mashup more robust during mashup creation and editing.

For example, consider the travel plan scenario from the introduction. With UWEM, the conference details widget emits the location data ET before the hotel search map widget is added. The hotel search map widget is preconfigured to read ETs with the "location" topic, and uses *monitorMulticast*() to get and show the corresponding map immediately. This means the map widget is plugged into the mashup application automatically and reacts to the available ETs, as soon as it is added to the page. No manual configuration is necessary for the wiring. If a widget cannot be wired automatically, users still have the option to configure it manually.

EVALUATION

We implemented an extended TS as a platform-independent JavaScript library that can be used in any web mashup framework, and integrated it with IBM Mashup Center (IBM MC) by extending its widget specification. The UWEM should simplify the widget communication configuration on this mashup platform.

Hypotheses

UWEM provides full routing schemes support for mashup platforms, and in most cases, widgets are automatic wired with other widgets. Therefore, our first hypothesis is that building mashups on the *UWEM* platform requires fewer operations than on a *nonUWEM* platform:

$$H_{TaskOpNum}\text{: } TaskOpNum_{UWEM} < TaskOpNum_{nonUWEM}$$

Our second hypothesis is that building mashups on the *UWEM* platform requires a smaller completion time than

on a *nonUWEM* platform:

$$H_{TaskCompTime}: TaskCompTime_{UWEM} < TaskCompTime_{nonUWEM}$$

Experimental Set Up

To evaluate UWEM, we used IBM MC to create two test mashup scenarios (*travel plan* and *customer analysis*), and used the Keystroke-Level Model (KLM) (Card, 1983) to predict the task execution time from the scenarios. The test scenarios are realistic; they are in fact similar to showcase applications on the IBM MC website (but they are in no way specific to IBM MC).

We created an analysis of the task with the KLM model and we performed an empirical study with twelve participants, where a comparison of the task completion times between building mashups with and without UWEM was made.

The first scenario is a travel plan mashup application. It is similar simpler than the example in the Introduction section. The scenario includes three widgets on the canvas. They are *Destination InputBox Widget*, *Weather Widget*, and *Map Widget*. In the scenario, *Destination InputBox Widget* connects the other two widgets and sends related data to the others.

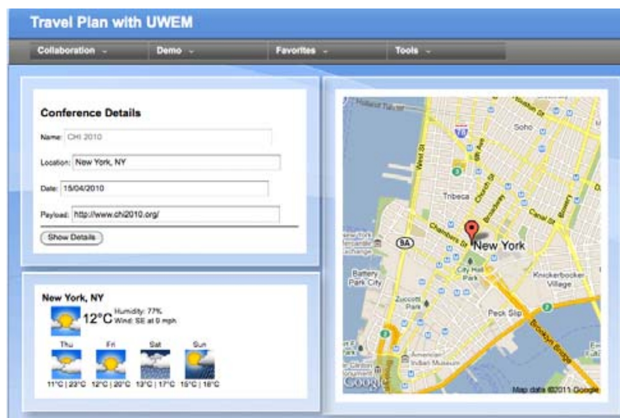

Figure 8. Travel plan mashup

The second test scenario is a mashup application for salesmen to do customer analysis. The scenario includes five widgets on the canvas. They are *Customer List Widget*, *Stock Widget*, *Map Widget*, *Website Displayer Widget*, and *Weather Widget*. In this scenario, *Customer List Widget* connects to the other four widgets and sends data to them.

The participants first do the scenarios on the normal IBM MC, and then they do the same task on the extended version of IBM MC using UWEM. Because the result of the KLM prediction might vary between expert users and normal users, we removed the time of the mental act of routine thinking operations in KLM. We took recommended operation times (Kieras, 2001) for each operation in the KLM prediction. For realistic testing, each participant was trained to use IBM MC before doing the tasks. The *task completion time*, *mouse click number*, *keystroke number*, and *mouse track length* were recorded using a Firefox extension.

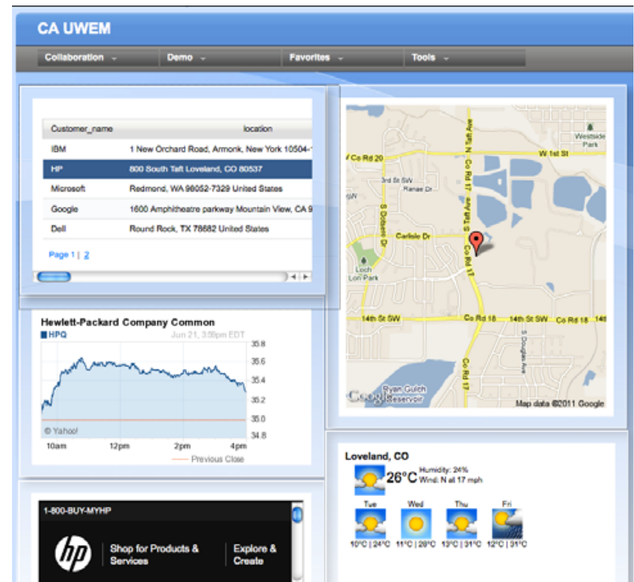

Figure 9. Customer analysis mashup

Each participant filled out two questionnaires, one before and one after the experimental tasks. In order to measure demographics, participants were asked to declare their gender, age and occupation in the pre-questionnaire. The main body of the pre-questionnaire consisted of 3 Likert-scale items, and the post-questionnaire consisted of 5 Likert-scale items and two open questions. The Likert-scale items were using a 7-point scale with standard labels, with a value range from 1 for “strongly disagree” to 7 for “strongly agree”. Participants had to rate their agreement with the following statements:

Pre-questionnaire:

1. I often use computers in my everyday life.
2. I frequently use mashup builder to build mashup applications.
3. I wire widgets every time when I am building mashup applications.

Post-questionnaire:

4. I understand the tasks.
5. I enjoyed using mashup builder to build applications.
6. It was easy to perform the tasks on the original IBM Mashup Center.
7. It was easy to perform the tasks on UWEM support IBM Mashup Center.
8. I prefer to use UWEM support IBM Mashup Center rather than the original one.

Item 1 measures the frequency of computer use. Item 2 and 3 measure the frequency of mashup builder use. Item 4 measures the validity of the results. Item 6 and 7 measure the perceived usability of UWEM in IBM MC. The two open questions asked the participants what they liked about UWEM, and what they did not like about UWEM (with a note to give suggestions for improvement).

Prediction with Keystroke-Level Model

Building mashups is a complex task. There are several common tasks involved in building mashups. These tasks are as follows (all these tasks are platform dependent, which means the prediction might vary on different platforms):

- **Drag and drop (D&D)** widget from drawer to canvas
- **Wire** two widgets

The KLM predictions for these two tasks are:

- $T_{dnd} = \mathbf{P} + 2\times\mathbf{B} = 1.1 + 2 \times 0.1 = 1.3$sec
- $T_{wire} = 3\times\mathbf{P} + 8\times\mathbf{BB} = 3\times1.1 + 8\times0.1 = 4.1$sec

All actions use the recommended average time in KLM. For the first scenario, the action sequence is as follows:

1. Point to the widget drawer **P**
2. Click widget drawer **BB**
3. Point to the Location widget **P**
4. Drag and drop Location widget from the widget drawer to the canvas **D&D**
5. Point to the widget drawer **P**
6. Click widget drawer **BB**
7. Point to the Map widget **P**
8. Drag and drop the Map widget from the widget drawer to the canvas **D&D**
9. Point to the widget drawer **P**
10. Click widget drawer **BB**
11. Point to the Weather forecast widget **P**
12. Drag and drop the Weather forecast widget from the widget drawer to the canvas **D&D**
13. Wire the Location widget with the Weather forecast widget **Wire**
14. Wire the Location widget with the Map widget **Wire**
15. Point to the input box in Location widget **P**
16. Input the location in the Location widget **T**(12) = 12 **K**
17. Point to the "Show Details" button **P**
18. Click "Show Details" button **BB**

$$T_{total_nonUWEM} = 8\mathbf{P} + 8\mathbf{B} + 12\mathbf{K} + 3\mathbf{D\&D} + 2\mathbf{Wire}$$
$$= 8\times1.1 + 8\times 0.1 + 12\times 0.28 + 3\times1.3 + 2\times4.1 sec$$
$$= 25.06\ sec$$

On the extended version of IBM MC, the action sequence to build the scenario is the same as the action sequence above except step 13 and 14 are omitted. The total prediction time for this action sequence is:

$$T_{total_UWEM} = 8\mathbf{P} + 8\mathbf{B} + 12\mathbf{K} + 3\mathbf{D\&D}$$
$$= 8\times 1.1 + 8\times 0.1 + 12\times 0.28 + 3\times1.3\ sec$$
$$= 16.86\ sec$$

The same analysis of prediction is made for scenario 2. The completion time of *nonUWEM* case is 36.2 seconds, compared with 19.8 seconds for *UWEM* case.

Demographics

There were 12 participants in the study, with a gender distribution of 1 female, 11 males. The age ranged from 22 to 42, with a median of 28.5. All participants were frequent computer users, as measured by item 1 of the questionnaire. The responses for this item ranged from 1 to 4 (on a scale of 1 to 7), with a median of 4 and an average of 6.9. None of the participants had used mashup builder before.

The participants were recruited in the campus. As a result, all participants were students. The labs are located in the Science Faculty, and hence most participants were studying a science related subject.

Results

Table 1 and 2 shows the results for the dependent variables as measured during the experimental tasks. The test probabilities for the differences between the *UWEM* and *nonUWEM* conditions are shown in the column P_{diff}.

Variable	***Average***	***Std. Dev***	P_{diff}
$MouseClicks_{nonUWEM}$	23.4	6.2	
$MouseClicks_{UWEM}$	10.1	3.2	0.0004**
$MouseTrackLen_{nonUWEM}$	9358	4287	
$MouseTrackLen_{UWEM}$	3000	2884	0.001**
$CompletionTime_{nonUWEM}$	75.4	33.6	
$CompletionTime_{UWEM}$	23.6	12.8	0.001**

Table 1. Descriptive statistics and test results for task 1

The table 2 shows the results for task 2.

Variable	***Average***	***Std. Dev***	P_{diff}
$MouseClicks_{nonUWEM}$	34.8	14.7	
$MouseClicks_{UWEM}$	16.5	6.9	0.000976*
$MouseTrackLen_{nonUWEM}$	18913	11583	
$MouseTrackLen_{UWEM}$	8900	10766	0.001953*
$CompletionTime_{nonUWEM}$	112.4	54.2	
$CompletionTime_{UWEM}$	42.4	38.2	0.000976*

Table 2. Descriptive statistics and test results task 2

The results show that the completion time for *UWEM* is significantly smaller (on average by 52%) than for the *nonUWEM* condition, so that $H_{TaskCompTime}$ can be accepted. Also the mouse click times and mouse track length for *UWEM* is significantly smaller (on average by 62% and 52%) than for the *nonUWEM* condition, so $H_{TaskOpNum}$ can be accepted as well.

Question	1	2	3	4	5	6	7	8
Avg. rating	6.9	1.2	1.8	6.4	5.8	4.3	5.8	6.1
Std. Deviation	0.3	0.4	1.7	1.7	1.6	1.9	1.8	1.5

Table 3. Descriptive statistics for the questionnaire.

Table 3 shows the results of the Likert-scale items of the questionnaire.

The responses to the open questions of the questionnaire were analyzed by performing a frequency count of sufficiently equivalent answers. One participant did not fill in the open questions section. 11 participants made positive comments about UWEM. The three most common positive comments about UWEM were:

1. No complex configuration is needed to setup communication among widgets.
   (5 participants, 41%)
2. Automatic wiring makes the task much easier.
   (5 participants, 41%)
3. It saves time to build mashups.
   (2 participants, 16%)

Other positive comments include very intuitive, No need to remember all links that requires to wire.

7 participants made negative comments about UWEM. The two most common negative comments about UWEM were:

1. No choice in case of ambiguities in wiring.
   (5 participants, 42%)
2. It would be better to perform the wiring while visually showing which fields are being connected.
   (3 participants, 28%)

The most common suggestion with regard to comment 1 was to make wiring configuration visible if there are more than one event sources are available.

Discussion

By applying KLM to the scenario, we obtained the prediction that the number of mouse clicks with *UWEM* should be less than half the number of *nonUWEM*, and the total time in the UWEM scenario should be about half the time in the *nonUWEM* scenario. This is of course mainly due to the fact that the *wire* tasks are not needed in the *UWEM* scenario. The empirical study produced a promising match with these predictions.

Many participants mentioned that the auto wiring makes the mashup platform less configurable to handle some complex mashup scenarios, which contains multiple event senders and receivers in one event. However, the UWEM doesn't replace but coexist with the existing widget event models. If the auto wiring among widgets is not possible, the user can still choose the existing wiring approach. Furthermore, the UWEM also provides a chance for the mashup user to explicitly edit the wiring of the widgets.

CONCLUSIONS AND FUTURE WORK

As event-based inter-widget communication is increasingly gaining attention, we addressed the issue that the strict event model does not fit mashup application well. Event models have a major impact on the flexibility and usability. Therefore, we introduced Linda and Tuplespace in mashups to provide a flexible event model to specify the existing message passing patterns and to extend the current mashup builder capability. Overall, UWEM can deliver a simpler plug and play experience of building mashup application for users.

The empirical study has given green light for a larger study of the efficiency of UWEM. For the larger study we are improving the instrumentation of the usability test setup so as to record the individual operations in the KLM model.

UWEM also has some issues that need to be addressed in future works:

- **Data Integration and Normalization**. The main challenge in building mashups is data integration and normalization. UWEM provides a chance to do data normalization within its TS.
- **Transaction Management**. In many circumstances, inter-widget communication is transactional. If the receiver fails in processing the data, the whole transaction can be rolled back. So the tuple in UWEM will be re-written into the TS if the taker widget fails.

REFERENCES

Abley, J., Lindqvist, K. and Abley, J. RFC 4786: Operation of Anycast Services. IETF, 2006.

Card, K., Thomas, T. P., and Newall, A. The psychology of human-computer interaction. Lawrence Erlbaum Associates, 1983.

Carrier et al. The business case for enterprise mashups. IBM White Paper, 2008.

Casquero et al. iGoogle and gadgets as a platform for integrating institutional and external services. 1st Workshop on Mash-Up Personal Learning Environments (MUPPLE), 2008, 37–41.

Chakravarthy, S. and Mishra, D. Snoop: an expressive event specification language for active databases. Data & Knowledge Engineering, 14(1):126, 1994.

Crockford, D. The application/JSON media type for JavaScript object notation (JSON), 2006.

Cugola, G. and Margara, A. Processing flows of information: from data stream to complex event processing. ACM Computing Surveys, 2011.

Cugola, G., Nitto, E. D., and Fuggetta, A. The JEDI event- based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering, 27, 9 (2002), 827–850.

Eugster, P. T., and Guerraoui, R. Content-based publish/subscribe with structural reflection. 6th USENIX Conference on Object-Oriented Technologies and Systems, 2001.

Gelernter, D. Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7, 1 (1985), 80–112.

Kaar, C. An introduction to widgets with particular emphasis on mobile widgets. Computing, Oct. 2007.

Ketter et al. Introducing an agile method for enterprise mash-up component development. IEEE Conference on Commerce and Enterprise Computing, 2009, 293–300.

Kieras, D. Using the keystroke-level model to estimate execution times, University of Michigan, 2001.

Ma, C. and Bacon, J. COBEA: a CORBA-based event architecture. 4th USENIX Conference on Object-Oriented Technologies and Systems, 1998.

Medhi, D. and Ramasamy, K. Network routing: algorithms, protocols, and architectures. Morgan Kaufmann, 2007.

Namoun, A., Nestler, T. and Angeli, A. Conceptual and usability issues in the composable web of software services. Current Trends in Web Engineering, vol. 6385, Springer, 396-407.

OpenAjax Alliance, OpenAjax hub 2.0 specification, 2009.

Ramalho, M. Intra-and inter-domain multicast routing protocols: a survey and taxonomy. Communications Surveys & Tutorials, IEEE, 3(1):225, 2000.

Rosenblum, D. S. and Wolf, A.L. A design framework for internet-scale event observation and notification, 6th European Software Engineering Conf. (ESEC/FSE). LNCS 1301. Springer, 1997.

Rozsnyai, S., Schiefer, J., and Schatten, A. Concepts and models for typing events for event-based systems. Inaugural International Conference on Distributed Event-based Systems, ACM, 2007, 62–70.

Sire, S., Paquier, M., Vagner, A., and Bogaerts, J. A messaging API for inter-widgets communication. 18th International Conference on World Wide Web, ACM, 2009, 1115–1116.

Tran, P., Gosper, J. and Yu, A. JXTA and TIBCO rendezvous – an architectural and performance comparison, 2003.

Wickramasinghe et al. BISSA: empowering web gadget communication with tuple spaces. Gateway Computing Environments Workshop (GCE), 2010, 1-8.

Wu, X., and Krishnaswamy, V. Widgetizing communication services. IEEE International Conference on Communications (ICC), 2010, 1–5.