Raven: Extending HTML for Peer-to-peer Synchronous Applications

Harry Chesley
Sean Kelly
Greg Kimberly
Tim Regan (Contact)

January 7, 2002

Technical Report
MSR-TR-2002-03

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Raven: Extending HTML for Peer-to-peer Synchronous Applications

Harry Chesley, Sean Kelly, Greg Kimberly, Tim Regan
Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
E-mail: timregan@microsoft.com

ABSTRACT
The Raven architecture provides object replication for any object system that implements object properties and methods. In the current Windows Internet Explorer implementation, it is particularly tightly integrated with HTML, allowing simple, rapid development of multi-user applications in a web page. Raven uses an XML schema approach to describing the objects to be shared among clients. The XML schemas determine which properties on the objects should be replicated and which methods should be remoted. In addition to the overall system design, this paper shows details of two examples: a simple chat application and a Spacewar game. It also provides detailed descriptions of some of the internal algorithms used.

KEYWORDS: Multi-user applications, HTML, peer-to-peer.

INTRODUCTION

Using the Internet is largely a solitary process, at least in terms of synchronous interaction. With the exception of some game sites like Microsoft's Gaming Zone and Shockwave.com, and simple chat interactions like IRC, users search for and read information in isolation. Even those sites generally use non-HTML technologies – custom controls, Java, and Director – to implement their multi-user applications.

In the Social Computing Group at Microsoft Research, we study the social aspects of computer interaction. In part, this involves creating and performing user studies with multi-user synchronous applications. In order to easily and quickly develop prototype systems, we needed a system that would let us create such applications with a minimum of development overhead, and which could be used by developers with only web page expertise.

We developed a system code-named Raven that extends HTML to allow simple development of multi-user applications. This approach makes it easy for anyone familiar with HTML web-page creation to develop distributed social applications. By basing the system on HTML, we allow users to make use of their existing knowledge of web page user interface design. The only new element is connecting users together through the Raven component.

Raven makes developing multi-user application even simpler by employing a replicated object technique [4] that is much easier for users to understand and use than a traditional distributed communicating object approach. Conceptually, the same object exists on all of the cooperating machines. Raven replicates changes to and method calls on the object from the originating machine to the others.

Raven uses an XML schema system [1, 5, 6] for describing which aspects of objects should be replicated. In addition to telling the system what portions of the object should be replicated remotely when changed locally, it protects the local object from unwanted remote changes to aspects of the object that were not intended to be exposed. We have attempted to remain as close as possible to existing schema systems such as SOAP [3] for these purposes.

SYSTEM DESIGN

Objects in the Raven system are viewed as existing simultaneously on all participating machines as replicated objects. The task of Raven is to synchronize new participants, replicate property changes and remote method calls from one machine to another. What aspects of an object should be replicated and what methods remoted are described in Raven's XML type schema. Channels provide a means of limiting the communicating clients and specifying the type of communication required.

Replicated Objects

Considering objects on different machines to be the same is much easier for a naïve developer to grasp than the traditional model of distributed communicating objects. In that traditional model, each machine has its own independent collection of objects which communicate with each other using message sending or remote procedure calls. Designing and debugging this type of system requires the designer to mentally keep multiple active objects in his or her head simultaneously. This becomes even more difficult when you need to remember that each of these pieces is acting independently and simultaneously. The most difficult bugs to find in these systems generally have to do with asynchronous independent interactions between the communicating objects.

The replicated object model is particularly suited to HTML-based applications, where the identical page description is loaded onto each participating machine. Since the application description is the same, it's quite natural to the developer that the objects described in it are also the same. Changes made on one machine automatically appear on all machines, without the developer needing to consider the details of that process.

But despite the fact that identical descriptions can easily be used to create identical, connected objects on multiple machines, Raven also allows for implementations involving different objects, in two senses: objects with different functionality, and objects with different implementations.

Two different functional objects, with different behaviors, can be registered with Raven as the same object. This can be used, for example, so that one user may have control over an object and the ability to modify it, while the other users can only see the object and the effects of the modifications. Raven's type system allows the developer to describe the object in abstract terms, as a collection of properties and methods, independent of its operation.

Similarly, how an object is actually implemented on each participating machine is separate from Raven and the Raven type description of the object. Although the current Raven implementation runs only on Windows with Internet Explorer, it is possible to implement Raven clients in virtually any environment that supports objects with properties and methods.

Types

A Raven type description consists of XML that describes each object in terms of the properties and methods that make up the object. These properties and methods are not a complete description of the object – local implementations may have more properties and/or methods than are mentioned in its Raven type definition. Rather, they describe those aspects of the object that are shared with other Raven instances of the object.

The following is an example description of an HTML checkbox that exposes only the state of the checkbox:

```
<raven>
  <ravenType name="ravenCheckbox">
    <element name="checked" type="boolean"/>
  </ravenType>
</raven>
```

Similarly, the following type definition exposes a single method:

```
<raven>
  <ravenType name="totalScore">
    <method name="addToScore">
      <argument name="count" type="ui4"/>
    </method>
  </ravenType>
</raven>
```

As base types, Raven uses the same types as are available in SOAP, plus a set that are specific to HTML objects. In addition, Raven has a mechanism for dynamically extending the base types by supplying an object that performs three type-related functions for a new type: object creation, conversion from object to XML for synchronization, and conversion from XML to object.

Channels

Communication between different instances of an object takes place on a Raven Channel. At the top level, a Raven channel connects a set of clients together, but these top-level connections can have sub-channels as well. Raven objects are registered on a particular channel, which determines both who will hear of changes and method calls and how.

The primary purpose of a channel is to determine which clients should be told of property changes or passed method calls on shared objects. At the top level, this is a question of who is participating is the multi-user application. But the sub-channel facility allows Raven applications to compartmentalize communication so that not all instances of the application hear about all of the objects. This can be used to limit communication overhead to only those clients that need to know. It can also be used to separate out communication that some clients should not hear for security purposes.

A secondary function of a channel is to determine the communications characteristics to be used. Channels can be reliable or unreliable, and they can be serialized or randomly ordered. For most applications, reliable serialized communication is needed, or at least desired to keep application development simple. However, there are cases where applications can handle a much larger number of clients if some aspects of their communication are not reliable and/or not serialized. For example, in a spatial virtual world simulation, moment-to-moment position data need not be 100% reliable, but chat input does need to be.

Serialization is the process of ensuring that the same actions occur on objects in the same order on all clients.

Serialization helps to limit or eliminate the sort of race-condition bugs that are so hard to isolate and fix in multi-client applications.

Since the same object may have different aspects that require different channel characteristics, Raven allows the same object to be registered simultaneously on multiple channels, which different types on each channel.

Using Raven

In our Windows Internet Explorer implementation, Raven exists as an ActiveX control that can be embedded in a web page. In addition to invoking Raven directly via methods on the Raven control, this allows Raven access to the web document object model (DOM).

Objects can be registered or created programmatically with Raven. Registered objects are pre-existing objects that the application tells Raven about. Created objects are objects that Raven itself is asked to create.

In each case, four things are needed for each object:

1. An object.
2. A Raven type.
3. A channel.
4. A Raven ID.

Aspects of the object, as defined by the type, are replicated via the channel and according to the communications characteristics of the channel.

The Raven ID uniquely identifies the object on the channel. If two Raven clients register objects with the same ID, they are considered to be instances of the same object. If Raven is asked to create an object and is not given an ID, it will make up a unique ID and use it on all clients.

Once an object is registered with Raven, property settings can be replicated and method calls remoted. However, for Raven to do this, it must know when these events happen. Since Raven works with any type of local object, it cannot intercept the events from within the object. Instead, developers must perform the actions on a special, "wrapped," version of the object, as supplied by Raven. For example, to set the innerText property of a textSpan object, the develop uses the following:

```
Raven.Wrap(textSpan).innerText = "new text";
```

The wrapped object returned from the Raven.Wrap() call takes any property sets and method calls and performs them both locally and remotely on all other clients.

Note that in order to maintain consistent serialization across all of the clients, these actions are not performed immediately on the local client. They must be delayed to fit into the sequence of operations on the channel in the same order as on the other clients.

Extending HTML

Some Raven objects are created in a scripting language such as JavaScript or VBScript. Some Raven objects are more complex things like the Windows Media Events player. But many Raven objects are simple DHTML DOM objects.

To make using Raven with DHTML as simple as possible, we have included a way to automatically register DHTML elements with Raven by including the Raven type in the DHTML tag. For example, to register a checkbox using the Raven type previously discussed, the following is all that is required:

```
<input type="checkbox" ravenType="ravenCheckbox">
```

Thus, to make the simplest Raven multi-user application, a shared checkbox, all we need is:

```
<input type="checkbox" ravenType="ravenCheckbox"
  onClick="Raven.Wrap(this).checked = this.checked">
```

A ravenID attribute can be used to set the Raven ID of the object as well. However, by default, the HTML ID is used, and this is usually sufficient.

Raven comes with a number of predefined Raven types for commonly shared DHTML objects.

Security & Authentication

Raven provides two types of security, but specifically does not provide one other type or authentication.

The most important security feature of Raven is that it operates within the Internet Explorer "sandbox." It does not allow anything to happen that is not possible with a normal web page. Since one of the primary goals of Raven is distribution of social computing experiments, making it totally safe for users was a primary concern. Confining it to IE allows users to run applications based on it free from concern, and also allows us to provide Raven as an auto-install, safe-for-scripting control.

The type facility of Raven provides another type of security, protecting the internal integrity of Raven-based applications. Since no properties changes or method calls are allowed that are not specifically described in an object's Raven type, applications can be written without concern for unexpected remote interference or side-effects.

On the other hand, Raven does not provide any type of network encryption to keep data being transferred from one client to another private. Nor does it attempt to authenticate either the Raven user or messages coming from other clients. Since our initial application for Raven involves information applications and social computing user studies, this level of security and authentication were considered unnecessary. These features are prime candidates for a future version of Raven.

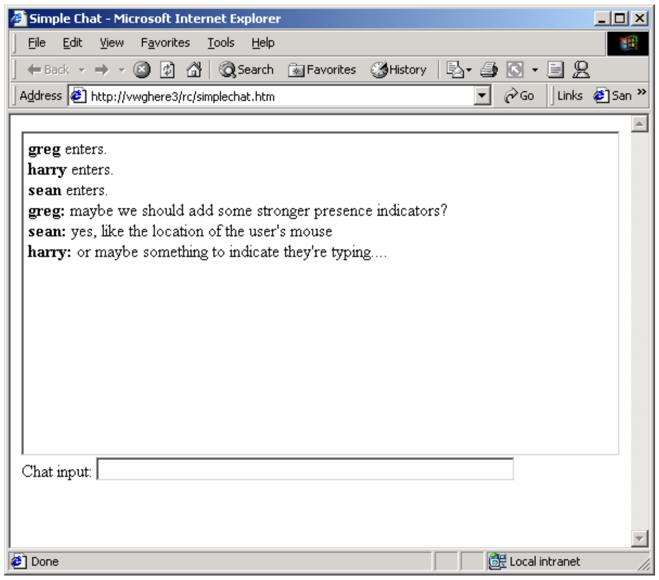
EXAMPLE APPLICATIONS

In order to test Raven, we developed a number of multi-user applications. We present two of them here. The intent is to show how applications are implemented in Raven, not to suggest that these particular applications are themselves of any special interest. The checkbox example was already shown above. Here we present SimpleChat and Spacewar.

SimpleChat

SimpleChat is a simple chat application in which each user sees a shared text history and can append comments to the end.

The following is a screen shot from SimpleChat:


Simple Chat - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://vwghere3/rc/simplechat.htm
Go Links San
Done
Local intranet

greg enters.
harry enters.
sean enters.
greg: maybe we should add some stronger presence indicators?
sean: yes, like the location of the user's mouse
harry: or maybe something to indicate they're typing....

Chat input:

The SimpleChat implementation consists of a <span> that contains all of the text history, embedded in a <div> that provides the scrollbars, plus one <div> for each user-entered comment. The history <span> is defined as:

```
<div style="height: 300px; border: 2px inset;
           padding: 4px; overflow: auto">
<span id="chatHistory" ravenType="chatHistorySpan">
</span>
</div>
```

It uses the Raven type chatHistorySpan, which is defined as:

```
<ravenType name="chatHistorySpan">
  <element name="innerHTML" type="string"/>
  <method name="insertAdjacentHTML">
    <argument name="where" type="string"/>
    <argument name="what" type="string"/>
  </method>
  <method name="scrollIntoView">
    <argument name="alignToTop" type="boolean"/>
  </method>
</ravenType>
```

When each instance of SimpleChat starts up, it asks the user for their nickname. From then on, new comments can be inserted in the chat history using the following JavaScript code:

```
var h = "<div><b>" + chatNickname + ":</b> " +
      "New input…" + "</div>";
Raven.Wrap(chatHistory).insertAdjacentHTML(
      "beforeEnd", h);
Raven.Wrap(chatHistory).scrollIntoView(false);
```

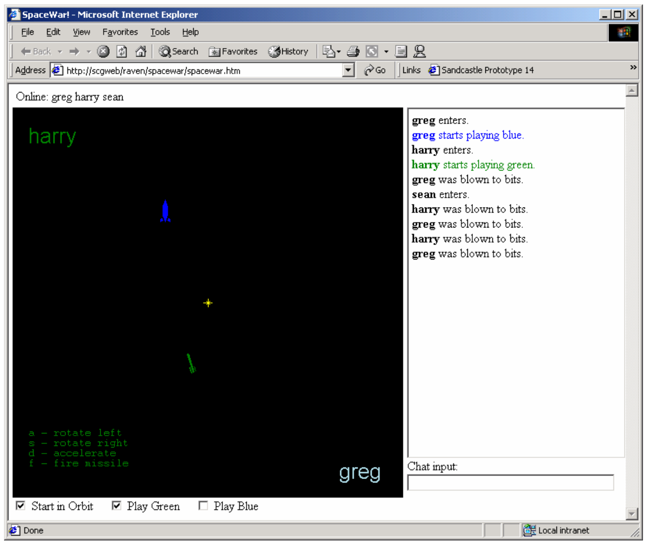
This inserts the new text into the history and scrolls the history to the bottom.

Note also that since the innerHTML property of the history span is declared as a property in the Raven type, new users histories will be synchronized to contain the preexisting history text, taken from other users already in the session.

Spacewar

In picking a test application for a new multi-user system, it seemed appropriate to reimplement what was the first multi-user computer application, Spacewar[2], though in the original it was played by two users sitting at the same keyboard. Spacewar was first implemented on a PDP-1 in 1961-1962. It can still be played today via a PDP-1 emulation written in Java. While not a complete recreation, Raven Spacewar does capture much of the feel of the original.

The following is a screen shot of Raven Spacewar:


SpaceWar! - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://scgweb/raven/spacewar/spacewar.htm
Go Links Sandcastle Prototype 14
harry
greg
Done
Local intranet

Online: greg harry sean

```
a - rotate left
s - rotate right
d - accelerate
f - fire missile
```

greg enters.
greg starts playing blue.
harry enters.
harry starts playing green.
greg was blown to bits.
sean enters.
harry was blown to bits.
greg was blown to bits.
harry was blown to bits.
greg was blown to bits.

Chat input:

☑ Start in Orbit ☑ Play Green ☐ Play Blue

Spacewar was intended to stress Raven's real-time performance. When all the missiles of a ship have been fired, the client controlling the ship does remote method invocations to set the positions of the ship and five missiles at each simulation step. The simulation speed was set to ten iterations per second, which was the maximum that JavaScript allowed on our 500MHz test machine. Adding half a dozen remote viewers, connected via Raven, had negligible impact on the simulation performance.

For Spacewar, we created JavaScript objects for each of the spaceships and missiles. These objects are registered with Raven when the session begins, so that every instance of Spacewar has the same ships and missiles. The Raven type for these is as follows:

```
<ravenType name="ship">
  <method name="setPosition">
    <argument name="x" type="r4"/>
    <argument name="y" type="r4"/>
  </method>
  <method name="setRotation">
    <argument name="r" type="r4"/>
  </method>
  <method name="setAnimation">
    <argument name="index" type="ui2"/>
  </method>
  <method name="setSound">
    <argument name="snd" type="string"/>
  </method>
</ravenType>

<ravenType name="missile">
  <method name="setPosition">
    <argument name="x" type="r4"/>
    <argument name="y" type="r4"/>
  </method>
  <method name="setSound">
    <argument name="snd" type="string"/>
  </method>
  <method name="show">
    <argument name="s" type="boolean"/>
  </method>
</ravenType>
```

The ship type exposes methods to set the position, rotation, animation, and sound. The missile type exposes methods to set the position, sound, and whether it is visible or not. The setPosition method, to take one example, is implemented as follows:

```
function setPosition(x, y)
{
  // Set the internal position
  this.x = x;
  this.y = y;

  // Set the visible position
  this.vect.style.pixelLeft =
      Math.round(x) - this.vect.style.pixelWidth/2;
  this.vect.style.pixelTop =
      Math.round(y) - this.vect.style.pixelHeight/2;
}
```

Each of the objects includes internal properties that keep the position as a real number. This is important for purposes of the simulation. But the actual visible ship or missile is displayed using a Vector Markup Language (VML) graphic. A reference to the graphic vector is stored in the object's vect property, and the style.pixelLeft/pixelTop properties of this are used to position the graphic element on the display.

The setPosition method is invoked remotely as follows:

```
Raven.Wrap(ship).setPosition(newX, newY);
```

Spacewar also includes a chat area, which is based on the SimpleChat application described earlier, with the addition of a list of current participants.

Further details of Spacewar are beyond the scope of this paper.

ALGORITHMS

The algorithms for most of Raven's services are apparent from its functional description. There are two areas, however, that are not as apparent: top-level channel coordination via an HTTP server, and serialization.

Client Rendezvous via HTTP

Clients can find each other in three ways with Raven: via a fixed address, via an HTTP directory service, and via an external directory service.

The first of these involves embedding the address of one of the participating machines in the HTML application. This is the simplest way for other clients to discover one of the other participating clients to rendezvous with. However, it has the limitation of requiring that one client always be running.

The last of these involves using some external directory service outside of Raven. This allows Raven to be open-ended with respect to top-level directory facilities, but doesn't provide an immediate answer for a developer needing a directory service.

The HTTP directory service is an ASP script designed to run on a Microsoft IIS HTTP server, and to talk directly with Raven. It could, however, easily be re-implemented in some other server-side technology. The algorithm that the ASP script uses is as follows:

1. A client requests the address of one of the active clients, via an HTTP request to the HTTP server.

2. If this is the first such request, the HTTP server tells the client to initiate a new session, and remembers the address of this client.

3. If a session is already active, the HTTP server returns the addresses of the previous clients that joined the session and remembers the address of

the new one. It passes the addresses of all clients since one or more may have left the session due to client or network failures.

4. While a client is active normally, it periodically contacts the HTTP server to let it know that it is still present. This allows the server to time-out clients that have crashed or become unreachable due to network issues.

5. When a client exits a sessions normally, it contacts the HTTP server again and the server removes its address from the list of active clients.

This algorithm is designed for an application where all clients connecting to the same web page are in the same session. For applications where multiple sessions are needed (such as many games), it is possible to implement a lobby in Raven that serves as a gateway to the application itself.

Serialization

One of the most important algorithms in Raven is the serialization algorithm. It is extremely important that this algorithm function reliably, both under normal circumstances and in the face of client and network failures. Serialization is one of the features of Raven that makes multi-user applications easy to develop. Without it, a multitude of timing issues can arise that complicate multi-user design and debugging.

The simplest serialization algorithm involves routing all messages through a single, serializing machine. This approach, however, has a number of problems. It places a large load on the serializing machine, and if the wrong machine is chosen as the serializer, it can seriously affect the performance of the application as a whole. Having a single serializer also goes against the overall peer-to-peer philosophy of Raven.

The algorithm used in Raven for serialization is fully peer-to-peer, and it easily recovers from unexpected client and network failures. In addition, it can be used to provide simulation timing support for an application that has time as well as serialization concerns.

Raven serialization thinks of the entire application as a distributed simulation and uses a global simulation clock to control and coordinate the process. Although the clocks are not required to be completely synchronized, they are generally quite close, within the variances introduced by network delays.

When a message is sent, either one that sets a property or one that invokes a remote method, it is stamped with a time that is far enough in the future of the simulation that it is expected that the message will arrive on all participating machines before that simulation time occurs. This is equivalent to saying that a message that is sent at time t is received at time t+x, even on the sending client.

Messages are processed in the time order stamped on them. If two messages have the same time, they are processed in the order established by the ID of the sender. This ensures identical serialization of messages on all clients.

Before a client can process a message at time t, it must know that it will not receive a message from another client at an earlier time. To know this, it must have received messages with time t or later from all clients. Clients that are not actively sending active messages send time marker messages periodically to allow the simulation to proceed.

The simulation clock advances continually, at the same rate as real time, so long as messages have been received from the other clients that allow it to. If it comes to a point such that messages are not received from all clients for least at the current simulation time, the simulation clock is suspended until messages are received that allow it to continue.

Stopping the simulation clock also prevents a client from sending messages. Because of this, once one client has suspended the simulation clock, before long all clients are suspended. This keeps the client's simulation clocks in lock-step within the time difference between the current time and the time stamped on outgoing messages.

Over time, the network delay to all clients may vary. The algorithm adjusts the message send/receive time stamp delta dynamically by comparing the time stamps on messages to the current simulation time when they are received. Raven attempts to keep this time difference as small as possible.

SUMMARY

Raven extends HTML to allow easy development of peer-to-peer, multi-user applications. By using object replication and peer-to-peer communication, Raven minimizes the intellectual hurdles to creating distributed applications. By integrating tightly with HTML, Raven makes developing such applications a simple and natural extension of normal HTML user interface development.

Raven was developed by the Social Computing Group in Microsoft Research in order to support rapid prototyping of social applications that allow people to interact over the Internet. To drive the development of Raven, we have created a number of simple applications that exercise Raven's distributed operation and HTML integration. Our next step is to create applications to support our research in social interaction via computers.

REFERENCES

1. *Extensible Markup Language (XML) 1.0* (http://www.w3.org/TR/1998/REC-xml-19980210).

2. Graetz, J.M., The Origin of Spacewar, Creative

Computing, August, 1981. Also http://spaceinvaders.retrogames.com/html/spacewar.html.

3. *Simple Object Access Protocol (SOAP) Version 1.1* (http://www.w3.org/TR/SOAP/).

4. Vellon, M., Marple, K., Mitchell, D. & Drucker. S. (1998). The Architecture of a Distributed Virtual Worlds System. *Proc. 4th Object-Oriented Technologies and Systems Conf.*

5. XML Schema Part 1: Structures (http://www.w3.org/TR/xmlschema-1/).

6. XML Schema Part 2: Datatypes (http://www.w3.org/TR/xmlschema-2/).