



[java.com.sun/javaone](http://java.com.sun/javaone)

## Using Comet to write a simple two player game

Jim Driscoll, Sr Engineer, Sun Micro

Jeanfrancois Arcand, Sr Engineer, Sun Micro

BOF-6584





Learn Comet by examining a toy (but functional!) program example.

GOAL



# Agenda

- Quick Introduction to Comet
- Tictactoe Demo
- Program Control Flow
- Grizzly Message Bus
- Summary

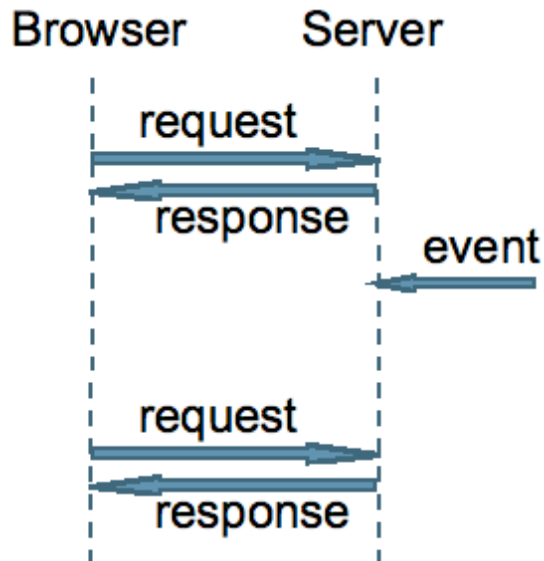
# What is Comet

- Since this is the last of 4 Comet talks today, we'll keep it short
- Different ways to organize program flow:
  - Page by Page
  - Ajax Polling
  - Ajax Push – Long Polling
  - Ajax Push – Streaming
- Comet Refers to both the Long Polling and Streaming methods of web programming

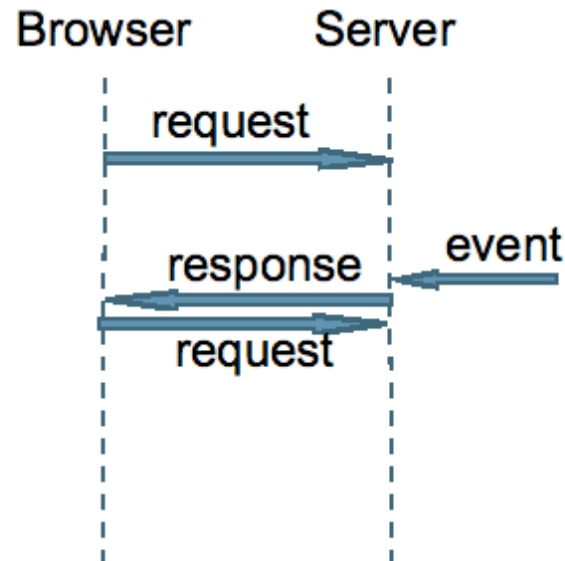


# A (very) quick Intro to Comet

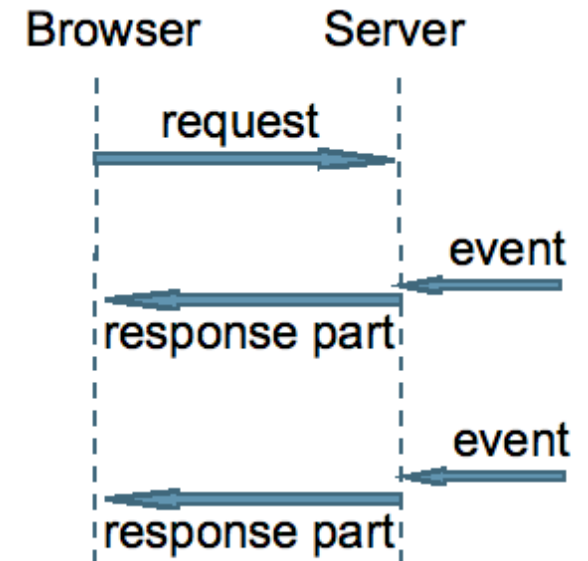
## Ajax (Polling)



## Ajax Push (Long Poll)



## Ajax Push (Streaming)



# Long Polling and Streaming

## ➤ Long Poll:

- Send a request to the server, wait for an event to happen, then send the response.
- The response is never empty.
- HTTP specification satisfied: indistinguishable from “slow” server
- Our demo uses this technique

## ➤ Http Streaming:

- Send a request, wait for events, stream multi-part/chunked response, and then wait for the events.
- The response is continually appended to.



# Agenda

- Quick Introduction to Comet
- [Tictactoe Demo](#)
- Program Control Flow
- Grizzly Messages Bus
- Summary

## Software used for the Demo

➤ Glassfish v3 tp2 (new release!)

- Needs to be enabled for Comet (domain.xml):
- `<http-listener id="http-listener-1" port="8080">`  
    `name="CometSupport" value="true" />`

`<property`

➤ Grizzly Comet APIs

- Not yet standardized
- Also runnable under Grizzly
- Concepts easily ported to Jetty, Tomcat, etc.

➤ Servlet APIs

➤ A browser Client, running Javascript

- Tested under FF3, FF2, IE7



# Tictactoe Demo

DEMO



# Agenda

- Quick Introduction to Comet
- Tictactoe Demo
- **Program Control Flow**
- Grizzly Messages Bus
- Summary

# Program Control Flow

- Initialize Comet
- Serve static page, connect to Comet servlet
- One player moves – Post is called
- On the server a Comet event is triggered
- On all clients, the update method is called
- And so on...



# Initialize Comet

```
public void init(ServletConfig config) throws
ServletException {
    super.init(config);

    ServletContext context = config.getServletContext
();
    contextPath = context.getContextPath() + "/"
TTTComet1";

    CometEngine engine = CometEngine.getEngine();
    CometContext cometContext = engine.register
(contextPath);
    cometContext.setExpirationDelay(120 * 1000);
}
```

# Initialize Comet

- Set everything up inside `Servlet.init()`
- Create a `CometContext`, register it
- This is where we can set up things like
  - Filters
  - Aggregators
  - Timeout

# Serve Static Page – ttt1.html (the html)

```
<iframe name="hidden" src="TTTComet1" frameborder="0"
height="0" width="100%" onload="restartPoll()"
onerror="restartPoll()" ></iframe>
<h1>Tic Tac Toe</h1>
<table>
<tr>
<td id="cell0"></td>
<td id="cell1"></td>
. . . .
</table>
<h2 id="gstatus">Starting to watch the game.</h2>
```

# Hidden iframe

- Set up the Hidden iframe
  - On first load, connects to server with a GET
    - Recommended technique for Comet is to use GET, not POST
  - Since it's 0 pixels high, and infinitely wide, it's hidden
  - Will render JavaScript as it comes in
- Perils of Hidden iframe technique
  - On some browsers (Safari, old IE) may not render until initial buffer full
  - Cannot check on progress via JavaScript, not suitable for large amounts of data.



# The Servlet Get (called by iframe)

```
protected void doGet(HttpServletRequest request,
HttpServletRequest response)
    throws ServletException, IOException {

    TTHandler handler = new TTHandler();
    handler.attach(response);
    CometEngine engine = CometEngine.getEngine();
    CometContext context = engine.getCometContext(contextPath);
    context.addCometHandler(handler);

}
```

# The initial GET

- Create a new CometHandler
  - (we'll look at that more in a minute)
- Attach the response to it
- Get the CometContext
- Attach the handler to the context
- The GET is now suspended, awaiting further action before a response is returned.



# Post is called (client)

```
<table>
<tr>
<td id="cell0"></td>
...
</table>

var url = "TTTComet1";
function postMe(arg) {
...
    var xhReq = new XMLHttpRequest();
    xhReq.open("POST", url, false);
    xhReq.setRequestHeader("Content-Type", "application/x-
www-form-urlencoded");
    xhReq.send("cell="+arg);
};
```



# Post is called (server)

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {

    String cellStr = request.getParameter("cell");
    PrintWriter writer = response.getWriter();

    .....
    game.turn(cell);
    .....
    CometEngine engine = CometEngine.getEngine();
    CometContext<?> context =
        engine.getCometContext(contextPath);
    context.notify(null);
}
```

# Something happens

- POST is called via a click on one client.
- We get the context again, and call `context.notify(null);`
- This sends a NOTIFY event to the handler.
- The handler will now update all listening clients
  - Sends a script which includes JSON formatted data
  - Closes all open connections when done.



# The TTTHandler

Inside the TTTComet class...

```
private class TTTHandler implements
CometHandler<HttpServletResponse> {

    private HttpServletResponse response;

    public void onEvent(CometEvent event) throws IOException
    {
        if (CometEvent.NOTIFY == event.getType()) {
            PrintWriter writer = response.getWriter();
            writer.write("<script type='text/
javascript'>parent.chImg(" + game.getJSON() + ")</script>
\n");
            writer.flush();
            event.getCometContext().resumeCometHandler(this);
        }
    }
}
```



# Sample JSON output

```
{ "win": "-1",  
  "board": ["0", "1", "10",  
            "0", "1", "10",  
            "1", "10", "0" ],  
  "turn": "10" }
```

# Client Side update

```
function chImg(args) {
var data = eval(args);

// redraw the board
for (i = 0; i < 9; i++) {
    document.getElementById("img"+i).src=
        "resources/"+data.board[i]+".gif";
}

// -1 is unfinished, 0 is tie, 1 is X win, 2 is O win
var statusMsg;
if (data.win == 0) {
    statusMsg = "It's a tie!";
} else if (data.win == 1) {
    ....
document.getElementById("gstatus").innerHTML = statusMsg;
```

# Client Update

- Using eval to parse the data
  - Only if you control both sides of the connection AND
  - Only if it either includes no user generated data, OR
  - If everything's been completely sanitized.
- Sanitization can happen in the Notification Handler OR
- Via a client side JSON library



## And reconnect to Comet Servlet (Get)

```
<iframe name="hidden" src="TTTComet1" frameborder="0"
height="0" width="100%" onload="restartPoll()"
onerror="restartPoll()" ></iframe>
```

....

```
var retries = 0;
function restartPoll() {
    if (retries++ > 10) {
        alert("The connection has too many errors - hit reload
to retry");
    } else {
        hidden.location = url;    // restart the
poll                                }
}
```



## Reconnecting to Comet

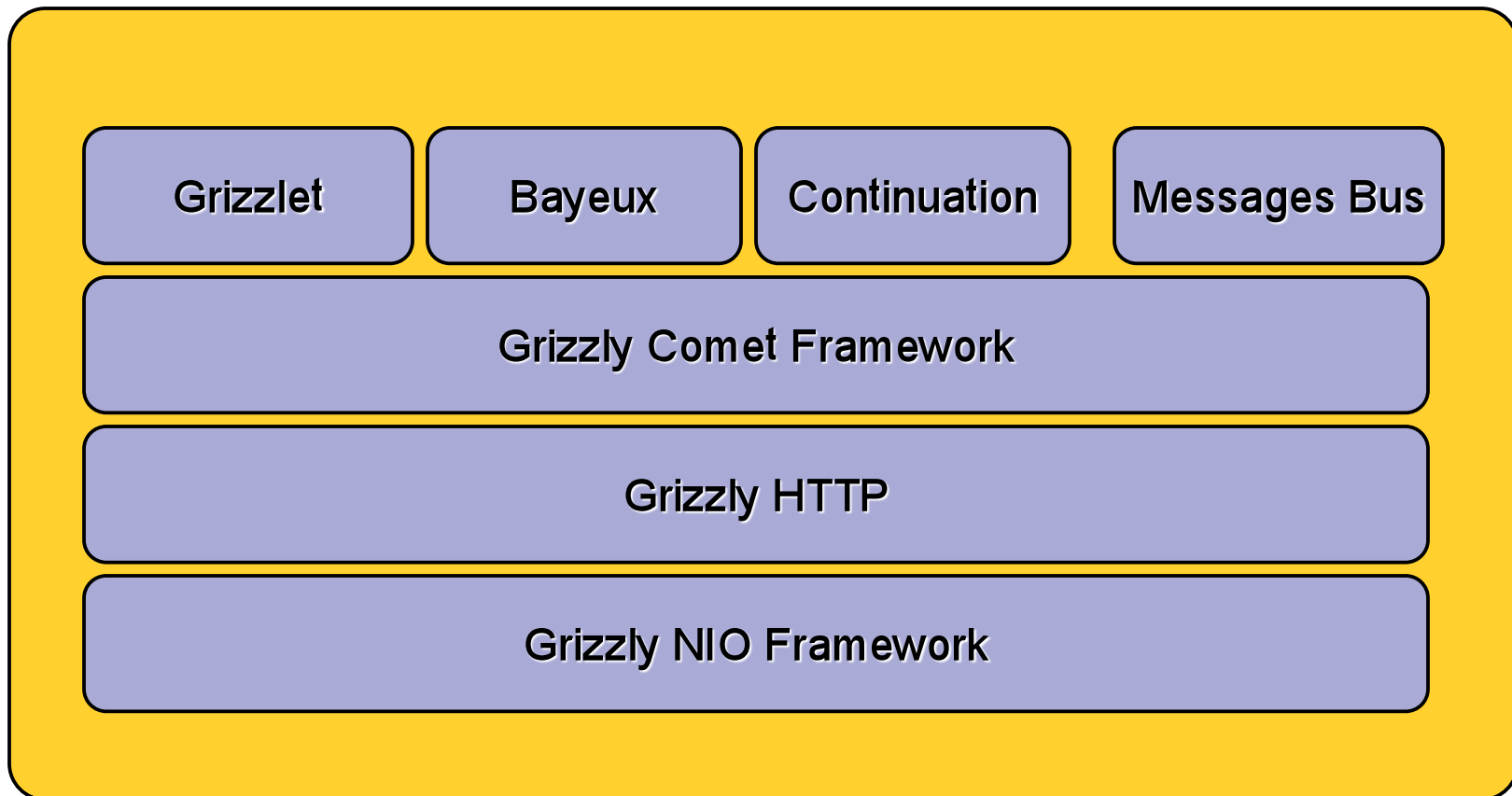
- Iframe onload is called when connection closed
- We include a counter to stop “client spin”
- All clients are now listening again for next event



# Agenda

- Quick Introduction to Comet
- Tictactoe Demo
- Program Control Flow
- **Grizzly Messages Bus**
- Summary

# Grizzly Comet Components



# Grizzly Messages Bus

- The Grizzly Messages Bus implements the Grizzly Comet Protocol (GCP).
- The GCP protocol is a very basic protocol that can be used by browser to share data, using the comet technique, between several clients without having to poll for it.
- The protocol is very simple. First, a client must subscribe to a topic:
  - `http://host:port/contextPath?subscribe=[topic name]&cometTechnique=[polling|log-polling|http-streaming]&message [text]`
- When issuing the URL above, the connection will be automatically suspended based on the cometTechnique specified

# Grizzly Messages Bus

- To share data between applications, a browser just need to send the following request:
  - `http://host:port/contextPath?publish=[topic name]&message=[text]`
- The Servlet can be used as it is or extended to add extra features like filtering messages, security, login, etc.
- Quite easy to write games using the Grizzly Messages Bus. No server side implementation required, just client side!



# Grizzly Messages Bus Demo

DEMO



# Agenda

- Quick Introduction to Comet
- Tictactoe Demo
- Program Control Flow
- Grizzly Messages Bus
- **Summary**



# Summary

## The Asynchronous Web Revolution is Now

- The Asynchronous Web will revolutionize human interaction
- Push can scale with Asynchronous Request Processing
- Writing Games is not that complicated.
- Writing Games using the Grizzly Messages Bus is even more easy!
- With GlassFish project and Project Grizzly, the revolution begins with your games today!



## For More Information

- TS-6482: Asynchronous Ajax for Revolutionary We Applications: Fri 12:50 (REPLAY)
- TS-4883: Java™ NIO Technology w/ Grizzly Framework: Fri 1:30
- <http://grizzly.dev.java.net>
- <http://weblogs.java.net/blog/jfarcand/>
- <http://weblogs.java.net/blog/driscoll/>

# THANK YOU



Jim Driscoll  
Jeanfrancois Arcand

