KAAZING

KAAZING WEBSOCKET GATEWAY

DETAILED TECHNICAL OVERVIEW

June 2010

Table of Contents

KAAZING WEBSOCKET GATEWAY

DETAILED TECHNICAL OVERVIEW

Kaazing's flagship product, Kaazing WebSocket Gateway, lets you build, deploy, and manage real-time web applications that can handle very large numbers of users and very high message volumes, using standard browsers and protocols, without changing the way you build and integrate software.

WHAT IS KAAZING WEBSOCKET GATEWAY?

Kaazing WebSocket Gateway is a platform for building web applications that require low latency, full-duplex communication between an end node, such as a browser or desktop, and a server. Using Kaazing WebSocket Gateway, web applications running within any browser can connect to real-time data over a reliable, bidirectional connection.

On the Web, real-time communication involves fast, event-driven, bidirectional, full-duplex message delivery between two or more machines.

At Kaazing WebSocket Gateway's core is Kaazing's patent-pending WebSocket Acceleration™ technology, which handles the transmission of hundreds of thousands of messages a second between clients and servers. WebSocket Acceleration can extend any TCP-based messaging format out to the web client, which turns any browser into a full-featured enterprise platform: a first-class citizen of any enterprise messaging system that is both fast and fully manageable.

WebSocket Acceleration requires no browser plug-ins, and integrates quickly and easily with enterprise messaging platforms. It does not rely on costly "polling" or "long-polling" (or "half-duplex") techniques that eat up valuable server resources, but is instead a true full-duplex communications model. What's more, it works with a wide variety of client-side technologies, including JavaScript, Adobe Flex (Flash), Microsoft Silverlight, and Java/JavaFX. Unlike traditional HTTP connections, a WebSocket is essentially a pure TCP socket across which any application data can flow in either direction at the same time. In short, it establishes a true bidirectional connection between the browser and the web server.

WebSocket Acceleration is based on the HTML5 WebSocket and Communication standards, which many industry experts and analysts believe to be the foundation of future web development. The web standards bodies (Web Hypertext Application Technology Working Group (WHATWG), World Wide Web Consortium (W3C), and Internet Engineering Task Force (IETF)) are now drafting the latest version of the HTML standard. Kaazing's active participation was critical in evolving the HTML5 WebSocket standard to what it is today. Our unique approach also allows older, pre-HTML5 browsers to benefit from the same HTML5 Communication features and to work with real-time messaging protocols without additional software.

Kaazing WebSocket Gateway can be used in a wide variety of industries. It is well suited to the high number of concurrent users common to telecommunications and social networking applications; to the demanding performance requirements of web-based trading, gaming, and auctions; and to the high message volumes of distributed applications and new media.

TRADITIONAL VERSUS WEBSOCKET ARCHITECTURE

To understand how Kaazing WebSocket Gateway and the WebSocket paradigm work, let's take a step back and look at how developers have tried to simulate real-time data delivery to web applications using a variety of Push techniques. Consider an application that needs to stream data from a back-end data feed (such as a stock ticker) to a client browser.

TRADITIONAL JAVA EE ARCHITECTURE WITH REAL-TIME PUSH SUPPORT

In a traditional architecture, such as the one shown in Figure 1, the conversation between the browser and the application server is completely separate from the conversation between the application server and the back-end data source. The application server isn't just forwarding traffic; it's actively involved in the translation, interpretation, and reformatting of every message flowing between the browser and the back-end.

For example, in a typical Java EE scenario:

- The browser connects to an application server via the Web.
- A custom Java EE Servlet using a Java Message Service (JMS) client retrieves data from the stock-trading client via a full-duplex TCP-based messaging protocol.
- The custom Java EE Servlet interprets custom JavaScript application protocols from the web application logic and maps these to standard APIs.
- Comet or reverse Ajax techniques might be used to push data to the clients.

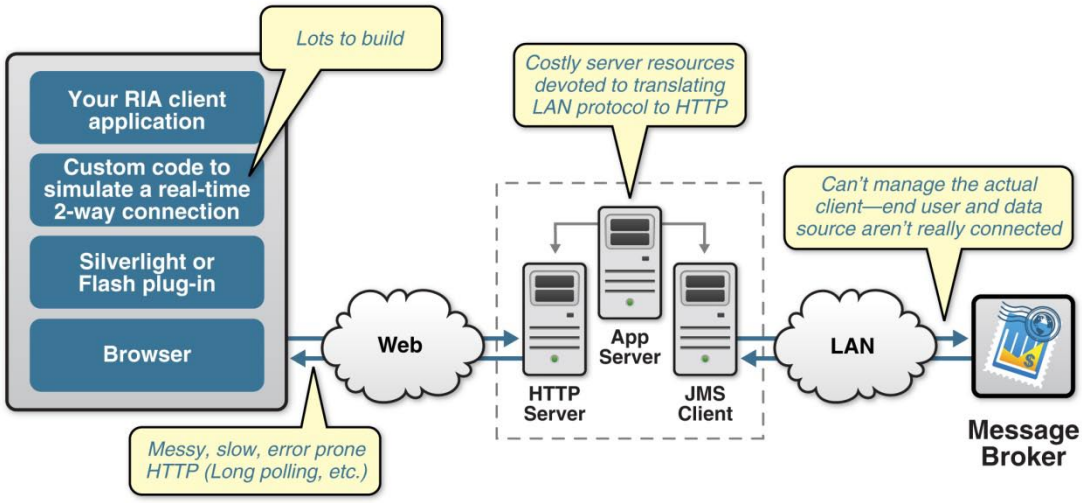

Figure 1—Example of web-based messaging with an intermediate Java EE container translating back-end traffic to browser HTML

Because the Java EE Servlet uses HTTP to talk to the browser, a complete full-duplex communication between the back-end message broker and the browser is never achieved: the Java EE Container is the throughput bottleneck.

There are additional issues with this implementation, namely:

- The developer must do the hard work of ensuring messages are delivered.
- The developer must manually implement a solution to push data to web clients that can bypass proxy servers and firewalls.
- Developers must design their own application protocol and map it into standard API calls.
- The JMS Client code in the Java EE Servlet must deal with many asynchronous requests, consuming considerable CPU resources.

In a traditional Java EE architecture, these issues apply whether the web application involved is for messaging, email, database access chat, or any other high-traffic, real-time protocol in which messages must flow reliably between browsers and back-end servers. Some example architectures are shown in Figure 2.

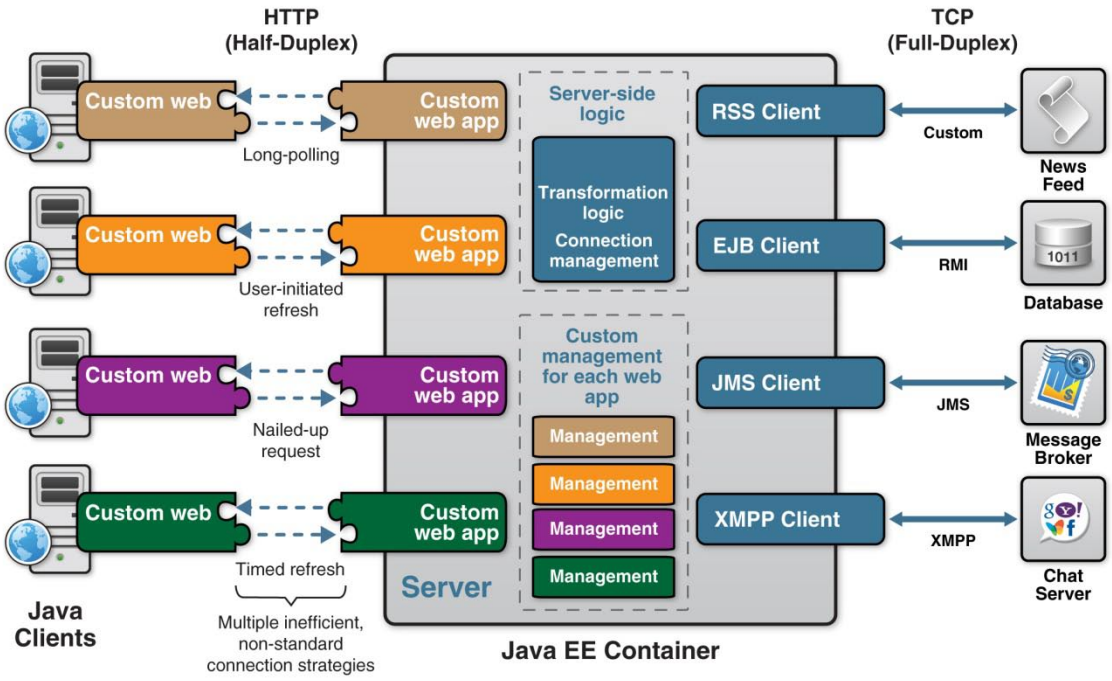

Figure 2—Example architectures in which a Java EE container connects browsers to back-end applications

POLLING, LONG-POLLING, AND STREAMING

Attempts to solve some of these issues tend to involve server-side push; that is, mechanisms that simulate server-initiated message delivery towards the client. The most notable of these is Comet or Reverse Ajax. Comet delays the completion of an HTTP response until the server has something to send to the client, a technique often called a "hanging-GET" or "pending-POST." Comet-based push is generally implemented in the client using JavaScript and uses the simulated server-side push strategies.

Polling is a technique in which the client (browser) sends HTTP requests at regular intervals and immediately receives a response, and represents the initial attempt to deliver real-time information on the Web. However, the downside of polling is that it generates a lot of unnecessary requests, which can impact both client and server performance.

In the case of long polling, or asynchronous polling, the browser client sends a request to the server and the server keeps the request open for a set period. If the server has something to send during that period, a response containing the message is sent to the client. If there is no data to send within the set time period, the server sends a response to terminate the open request, and the client then re-opens another.

In the case of streaming, the browser client sends a complete request and the server maintains an open response that is updated whenever it has a message ready to be sent, keeping the connection open to deliver future messages. Streaming, however, is still encapsulated in HTTP, so intervening HTTP proxy servers may choose to buffer the response, increasing the latency of the message delivery.

These methods (polling, long polling, and streaming) are inefficient for several reasons:

- In low-message-rate situations, polling results in many connections being opened and closed needlessly.
- In high-message rate situations, long polling results in a continuous loop of immediate polls.
- Communication still involves HTTP headers, which can contain hundreds or even thousands of bytes of additional, unnecessary data and often account for the majority of the network traffic.
- If secure connections (connections using Transport Layer Security (TLS)) are used in combination with long-polling, the setup and tear down of each connection can be even more taxing on server resources.

Simply put, HTTP wasn't designed for real-time, full-duplex, bidirectional messaging.

HTML5 WEBSOCKET TO THE RESCUE!

Defined in the Communications section of the HTML5 specification, HTML5 WebSocket represents the next evolution of web communications — a full duplex, bidirectional communications channel that operates through a single socket over the Web. HTML5 WebSocket provides a true standard that you can use to build scalable, real-time web applications. In addition, since it provides a socket that is native to the browser, it eliminates many of the problems to which Comet solutions are prone. HTML5 WebSocket removes the overhead and dramatically reduces complexity.

HTML5 WebSocket provides such a dramatic improvement from the old, convoluted "hacks" that are used to simulate a full-duplex connection in a browser, that it prompted Google's Ian Hickson, the HTML5 specification lead, to say:

> *"Reducing kilobytes of data to 2 bytes…and reducing latency from 150ms to 50ms is far more than marginal. In fact, these two factors alone are enough to make WebSocket seriously interesting to Google."*[1]

Let's look at how HTML5 WebSocket can offer such an incredibly dramatic reduction of unnecessary network traffic and latency by comparing it to conventional solutions.

POLLING VERSUS WEBSOCKET

To reveal the shocking amount of header overhead that is associated with each polling request, let's compare a polling application with a WebSocket application. Examples 1 and 2 show the HTTP header data for just a single request and response in a polling application.

Example 1—HTTP request header in a polling application

```
GET /PollingStock//PollingStock HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/PollingStock/
Cookie: showInheritedConstant=false; showInheritedProtectedConstant=false;
showInheritedProperty=false; showInheritedProtectedProperty=false;
showInheritedMethod=false; showInheritedProtectedMethod=false;
showInheritedEvent=false; showInheritedStyle=false;
showInheritedEffect=false
```

[1] IETF website: http://www.ietf.org/mail-archive/web/hybi/current/msg00784.html

Example 2—HTTP response header in a polling application

```
HTTP/1.x 200 OK
X-Powered-By: Servlet/2.5
Server: Sun Java System Application Server 9.1_02
Content-Type: text/html;charset=UTF-8
Content-Length: 21
Date: Sat, 07 Nov 2009 00:32:46 GMT
```

In these examples, the total HTTP request and response header information overhead contains 871 bytes and that does not even include any data. Of course, this is just an example, but the header data can in some cases exceed 2000 bytes. In this example application, the data for a typical stock topic message is only about 20 characters long. As you can see, this salient bit of information is effectively drowned out by the excessive and unnecessary header information.

So, what happens when you deploy this application to a large number of users? Let's take a look at the network throughput for just the HTTP request and response header data associated with this polling application in three different use cases.

- **Use case A**: 1,000 clients polling every second:
  Network throughput is (871 x 1,000) = 871,000 bytes = 6,968,000 bits per second (6.6 Mbps)
- **Use case B**: 10,000 clients polling every second:
  Network throughput is (871 x 10,000) = 8,710,000 bytes = 69,680,000 bits per second (66 Mbps)
- **Use case C**: 100,000 clients polling every 1 second:
  Network throughput is (871 x 100,000) = 87,100,000 bytes = 696,800,000 bits per second (665 Mbps)

These use cases illustrate an enormous amount of unnecessary network throughput. If only we could just get the essential data over the wire. You can do just that with HTML5 WebSocket. When you rebuild the same application using HTML5 WebSocket, each of these messages is contained in a WebSocket frame that has just two bytes of overhead (instead of 871). Frames can be sent full duplex in either direction at the same time. In the same web application example, the HTTP header overhead can be replaced with just two bytes, as shown in Example 3. Here, each frame of data starts with a 0x00 byte and ends with a 0xFF byte and contains UTF-8 data between them.

Example 3—WebSocket Frame

```
\x00Hello, WebSocket\0xff
```

Now, let's take a look at how using HTML5 WebSocket affects the network throughput overhead in the same three use cases for the sample web application.

- **Use case A**: 1,000 clients receive 1 message per second:
  Network throughput is (2 x 1,000) = 2,000 bytes = 16,000 bits per second (0.015 Mbps)
- **Use case B**: 10,000 clients receive 1 message per second:
  Network throughput is (2 x 10,000) = 20,000 bytes = 160,000 bits per second (0.153 Kbps)
- **Use case C**: 100,000 clients receive 1 message per second:
  Network throughput is (2 x 100,000) = 200,000 bytes = 1,600,000 bits per second (1.526 Kbps)

As you can see in Figure 3, HTML5 WebSocket provides a dramatic reduction of unnecessary network traffic compared to the polling solution.

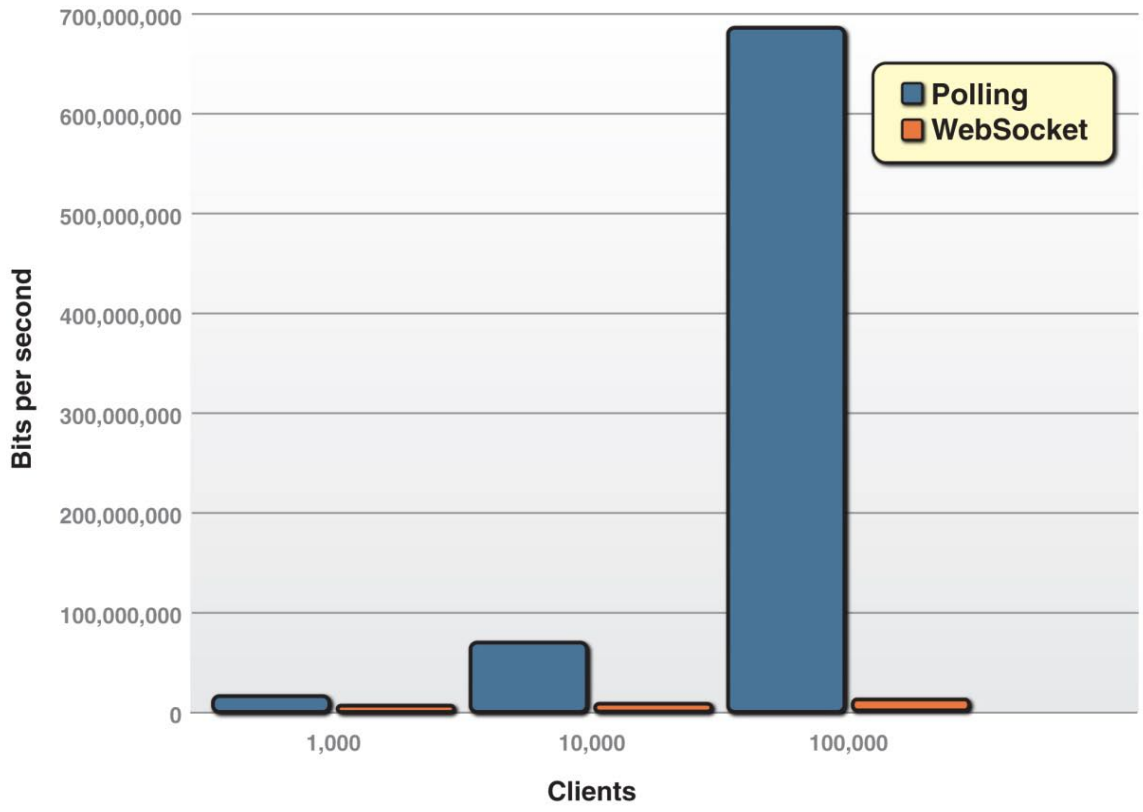

Figure 3—Comparison of the unnecessary network throughput overhead between the polling and the WebSocket applications

Additionally, HTML5 WebSocket provides a significant reduction in latency, as shown in Figure 4. In the top half of the figure, you can see the latency of the half-duplex polling solution. If we assume, for this example, that it takes 50 milliseconds (ms) for a message to travel from the server to the browser, then the polling application introduces a lot of extra latency, because a new request has to be sent to the server when the response is complete. This new request takes another 50ms and during this time the server cannot send any messages to the browser, resulting in additional server memory consumption.

In the bottom half of the figure, you see the reduction in latency provided by the WebSocket solution. Once the connection is upgraded to WebSocket, messages can flow from the server to the browser the moment they arrive. It still takes 50ms for messages to travel from the server to the browser, but the WebSocket connection remains open so there is no need to send another request to the server.

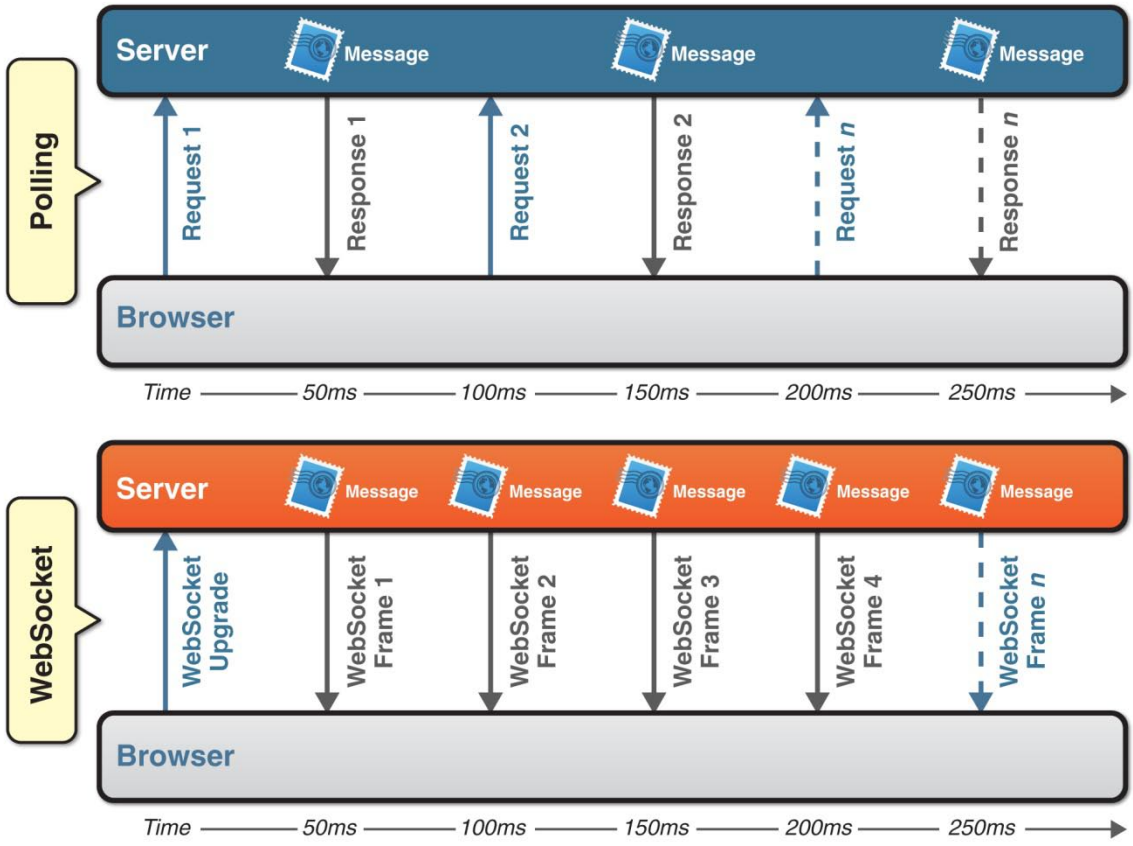

Figure 4—Latency comparison between the polling and WebSocket applications

KAAZING WEBSOCKET GATEWAY ARCHITECTURE

Kaazing WebSocket Gateway is an enterprise-grade HTML5 communication gateway that enables full-duplex communication from the browser to any TCP-based back-end service, such as JMS, JMX, IMAP, and Jabber servers. Kaazing WebSocket Gateway not only handles millions of messages reliably and efficiently, it also eliminates the need for complex server- and client-side architectures to bridge various protocols to the browser over HTTP.

But, what about browsers that don't support HTML5 WebSocket? Fortunately, Kaazing's unique architecture is designed to work seamlessly with older browsers, emulating WebSocket functionality using JavaScript as a last resort. Kaazing WebSocket Gateway not only supports the native WebSocket protocol, the Kaazing libraries also leverage all possible browser features to emulate WebSocket functionality in older browsers that do not support WebSocket. Kaazing's unique Opportunistic Optimization™ uses the best possible connection, regardless of whether clients and intermediate devices support the latest protocols.

Kaazing WebSocket Gateway allows developers to code directly against back-end services, using JavaScript or other technologies, such as Adobe Flex (Flash), Microsoft Silverlight, and Java/JavaFX, thus eliminating the need for complex server- and client-side architectures. As a result, Rich Internet Applications (RIAs) on the browser can communicate directly with the back-end data source, as shown in Figure 5.

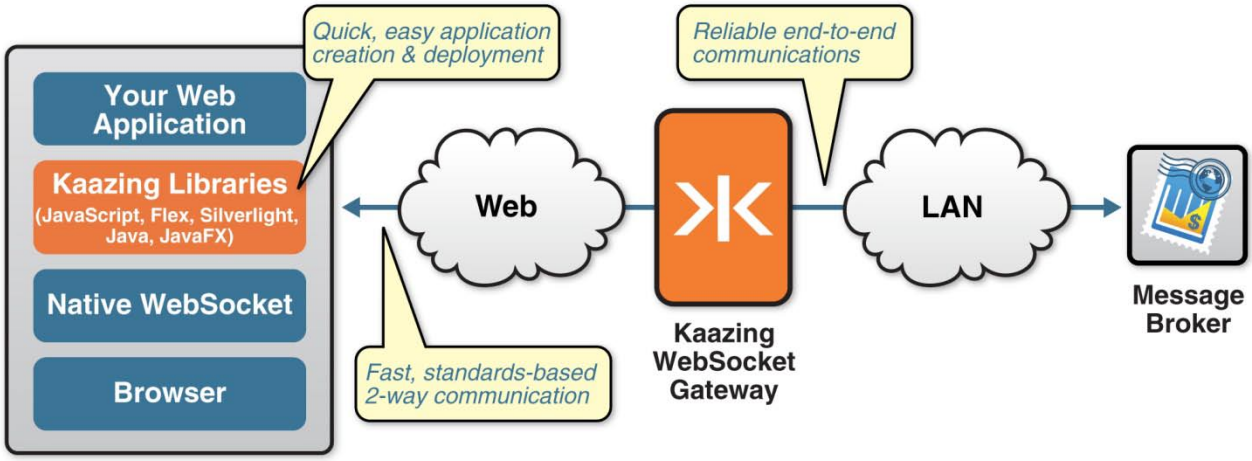

Figure 5—Simple, reliable, high-capacity communication between client and server with Kaazing WebSocket Gateway

To build an application, developers simply configure Kaazing WebSocket Gateway to communicate directly with the back-end APIs (for example, Stomp or XMPP), and develop their client web application, allowing them to maintain focus on what matters most: building applications, instead of reinventing the wheel.

HTML5 WEBSOCKET ARCHITECTURE

Kaazing WebSocket Gateway interprets both emulated and native TCP calls from clients, and passes them directly to any TCP-based back-end service, such as a JMS broker, a database, an IMAP server, or an instant messaging server. As Figure 6 illustrates, an end-to-end TCP connection between browser and back-end server over which messages are sent with minimal overhead.

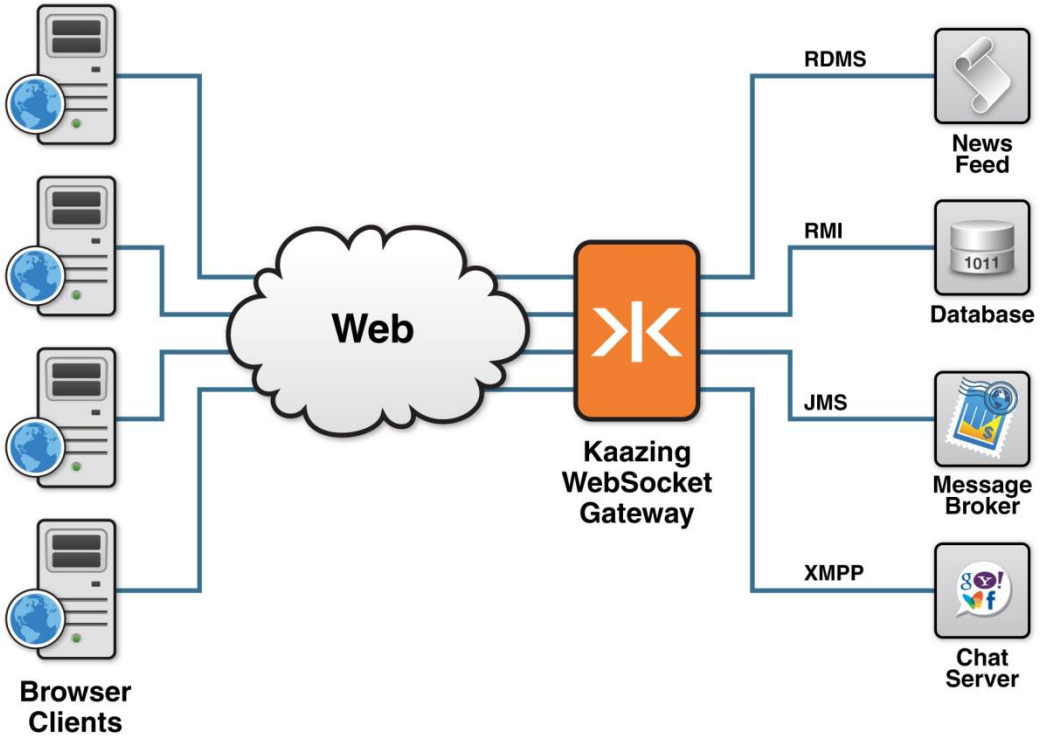

Figure 6—Kaazing WebSocket Gateway connects native back-end TCP calls to full-duplex browser client connections

Any browser can connect to Kaazing WebSocket Gateway. Modern, HTML5 WebSocket-enabled browsers natively support the calls required to establish a full-duplex, bidirectional communication channel with Kaazing WebSocket Gateway and, as a result, the back-end service. Kaazing also provides client libraries that are loaded into browsers that do not inherently support HTML5 WebSocket. These client libraries enable the browser client to emulate WebSocket behavior. In both cases, developers can use the HTML5 WebSocket APIs and there is no difference in the development process.

ENTERPRISE HTML5 WEBSOCKET ARCHITECTURE

Deploying Kaazing WebSocket Gateway to the enterprise is straightforward, because it can traverse the many firewalls and proxy servers that often separate enterprise networks from browser and desktop clients. In this respect, deployment resembles that of a traditional Application server. Kaazing WebSocket Gateway ensures that message delivery guarantees are maintained.

This approach also means that Kaazing WebSocket Gateway benefits from high-availability features in load-balancing gateways; it also allows configurations in which multiple tiers of Kaazing WebSocket Gateway are used to streamline message delivery.

Figure 7 shows a network in which multiple Kaazing WebSocket Gateways, front-ended by a reverse proxy server, are sending real-time data to browser clients. Kaazing WebSocket Gateway allows real-time, full-duplex communications on top of an existing networking infrastructure, and can co-exist with traditional web application servers.

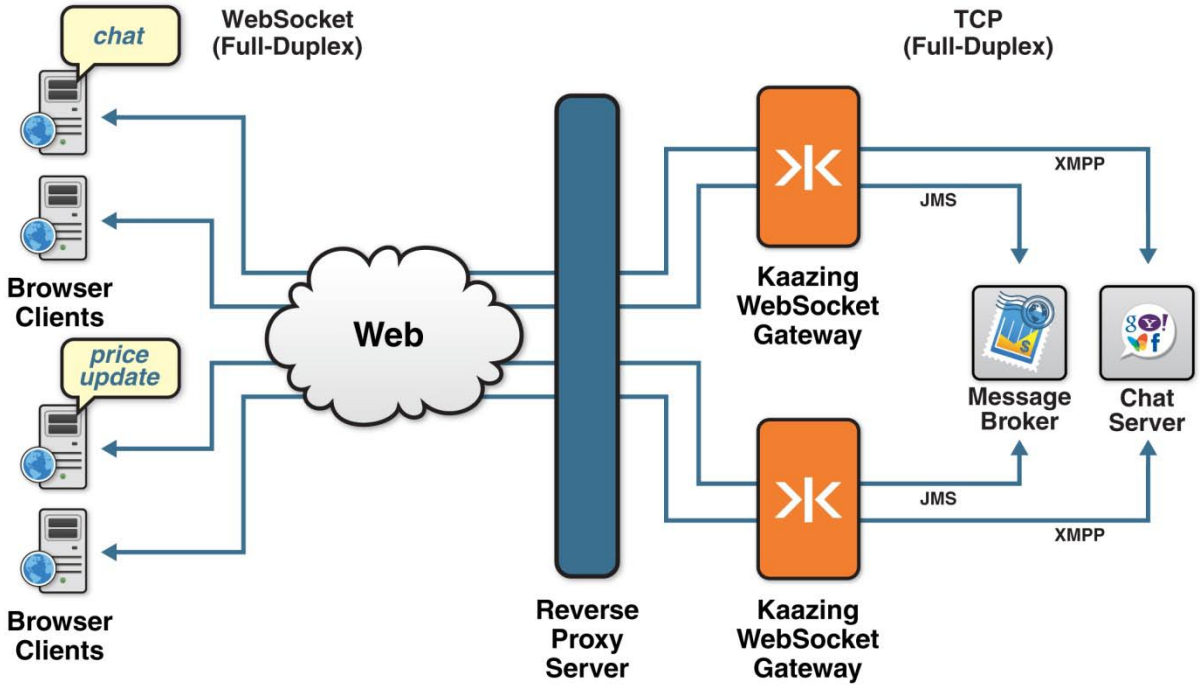

Figure 7—Multiple Kaazing WebSocket Gateways, front-ended by a reverse proxy server

HTML5 SERVER-SENT EVENTS ARCHITECTURE

Server-Sent Events (SSE) is a standard that describes how servers can initiate data transmission towards clients once an initial client connection has been established. These events are commonly used to send message updates or continuous data streams to a client. As such, SSE effectively standardizes server-side push (Comet) technology. SSE was designed to enhance native, cross-browser streaming through the new EventSource JavaScript API, through which a client requests a particular URL in order to receive an event stream.

Kaazing WebSocket Gateway supports broadcast of data from a back-end data source to all clients. For this, the back-end data source typically sends data to Kaazing WebSocket Gateway using the User Datagram Protocol (UDP) and Kaazing WebSocket Gateway then transmits new notifications to browsers using SSE. Due to its lossy nature, UDP is typically not very well suited for delivery outside of the firewall and doesn't work for point-to-point web connections. Kaazing WebSocket Gateway, however, is ideally suited for this task.

Figure 8 shows a news feed that broadcasts custom UDP messages to Kaazing WebSocket Gateway, which in turn fans the packet data out to all connected browser clients using SSE.

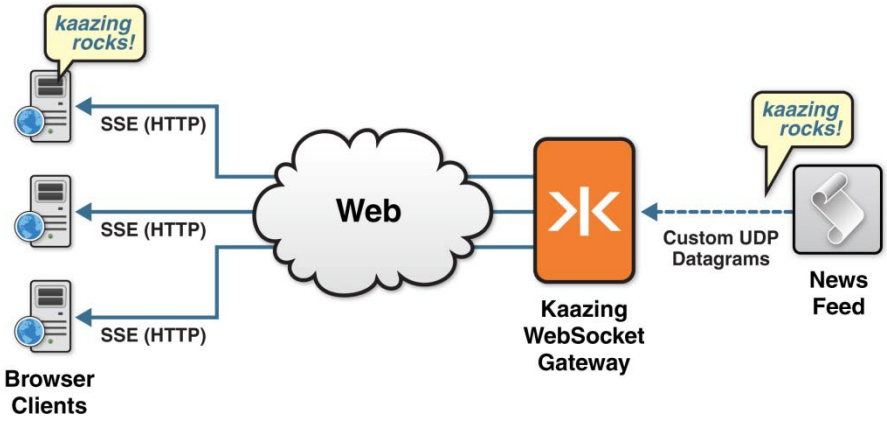

Figure 8—Kaazing WebSocket Gateway distributing a broadcast message to clients from a back-end news feed

USING MULTICAST FOR RELIABLE SERVER-SENT EVENTS (SSE) IN ENTERPRISE NETWORKS

Enterprise environments demand resilient, highly available deployment models that can fail over when a component of the messaging fabric fails. To accomplish this, Kaazing WebSocket Gateway supports multicast messages from back-end data sources.

In this configuration, the back-end data source transmits packets of data once to a multicast IP (an IP address with a multicast address mask in the range 224.0.0.0 to 239.255.255.255.). Multiple Kaazing WebSocket Gateways can listen to this multicast address and retrieve the messages without the sender having to be aware of each Kaazing WebSocket Gateway. This simplifies the deployment of redundant messaging systems within the network. It also reduces the workload of the back-end data source because it only needs to send each packet once to ensure every Kaazing WebSocket Gateway in a high-availability cluster receives it.

Figure 9 shows a news feed that uses multicast to send messages. Multiple Kaazing WebSocket Gateways, front-ended by a reverse proxy server or load balancing router, are configured to listen for data on the multicast address and fan the packet data out to all connected browser clients using SSE.

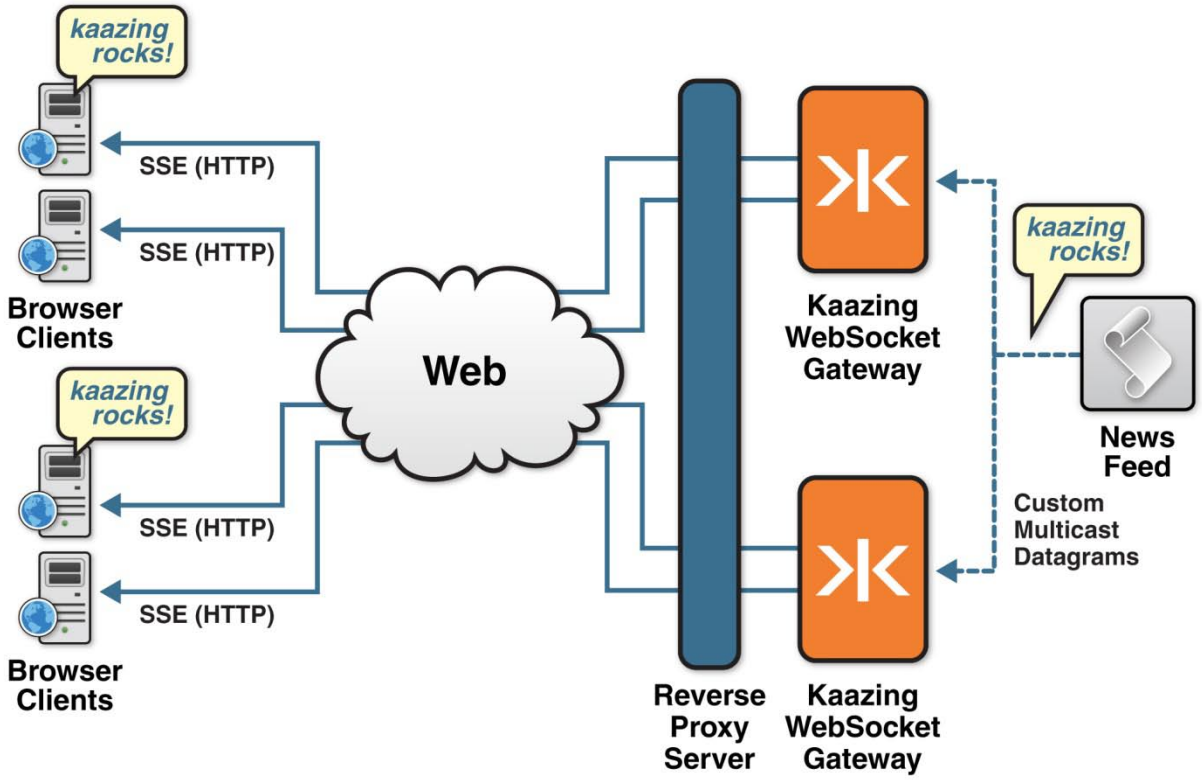

Figure 9—Using multicast for reliable distribution of messages in an enterprise configuration

This model scales architecturally, rather than through brute-force approaches like adding capacity, ensuring that Kaazing deployments can satisfy even the most demanding message delivery applications.

VIRTUAL PRIVATE CONNECTION FOR CLOUD COMMUNICATION

The Kaazing WebSocket Gateway can be configured to allow TCP (and UDP) clients to connect to servers over the web without the need for any special Kaazing or WebSocket libraries, thus creating a *virtual private connection* for machine-to-machine communication. Kaazing WebSocket Gateway was designed not only to proxy TCP protocols using the WebSocket protocol, it can also be run in a reverse mode of WebSocket protocol to TCP.

This design allows system and network administrators to configure two or more gateways so applications can traverse the web securely through firewalls and proxy servers. Enterprises and small companies alike can now deliver sophisticated server-to-server systems and rich client applications over a LAN or WAN web infrastructure in the same manner as conventional distributed applications, all without the expense or complexity of a private line.

Virtual private connection allows many types of applications to be constructed, such as real-time supply chain, inter-cloud and cloud-to-cloud communication, fixed-income exchanges, local office management, external cloud applications using data remaining within the enterprise, and so on. Figure 10 shows a legacy, distributed application that relies on having special ports opened on the corporate firewall.

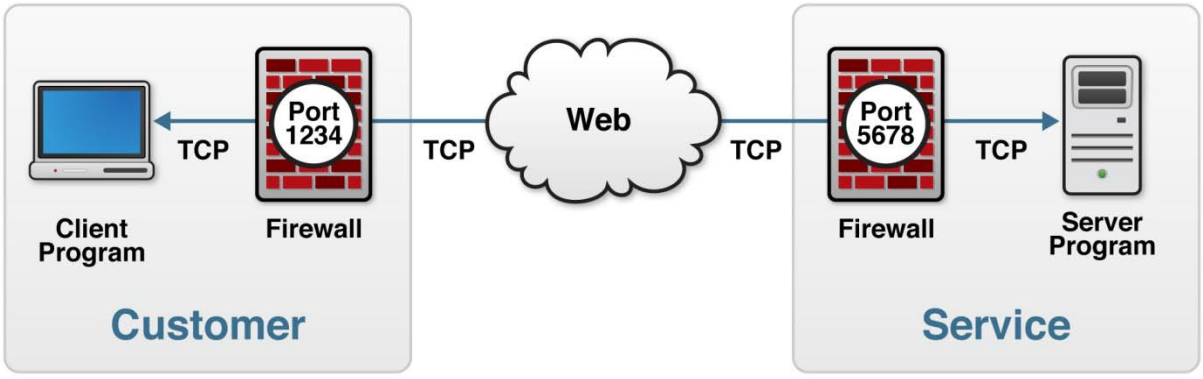

Figure 10—Legacy distributed application using special ports on the corporate firewall

These legacy applications can now be extended over the Web with TLS encryption, traverse firewalls and proxy servers, and retain all of its functionality.

To configure a virtual private connection you need at least two Kaazing WebSocket Gateways: one configured to proxy TCP traffic over WebSocket and another configured in reverse to accept WebSocket traffic and proxy it to a TCP-based back-end service or program. This way, you can set up communication between mutually exclusive networks, traverse proxy servers, and so on. Figure 11 shows an example virtual private connection topology in which TCP traffic from the client program on the Customer instance is proxied to the Service instance using the WebSocket protocol over port 80.

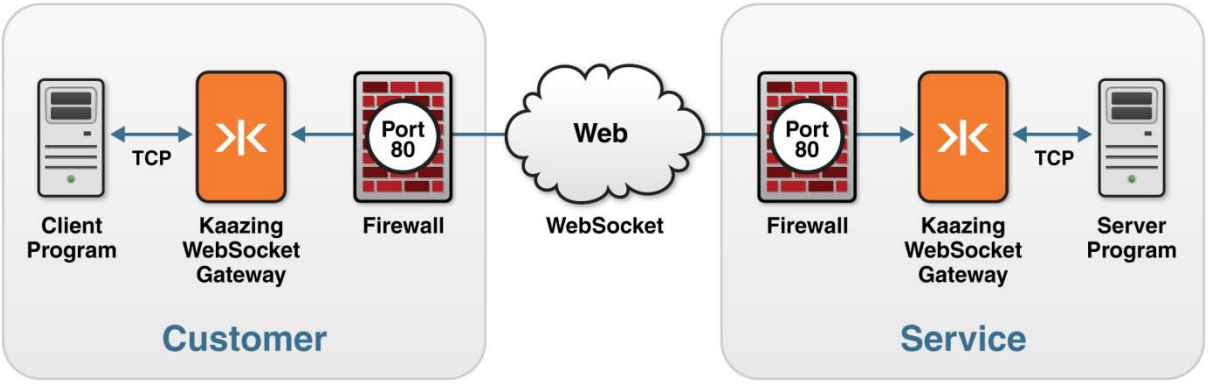

Figure 11—Kaazing Virtual Private Network topology over port 80

CONNECTION OFFLOADING, FANOUT, AND GLOBAL DEPLOYMENT

Kaazing WebSocket Gateway supports distributed architectures and WebSocket fanout through connection and subscription offloading. Message brokers and related services are typically not designed to handle tens or hundreds of thousands of concurrent connections, but Kaazing WebSocket Gateway is uniquely qualified to do so.

Kaazing WebSocket Gateway provides stateful services, which support connection and subscription offloading for the back-end messaging server to achieve a high level of scalability. Figure 12 shows an example fanout scenario in which gateways share connections to a back-end server instead of simply proxying connections to the back-end messaging service.

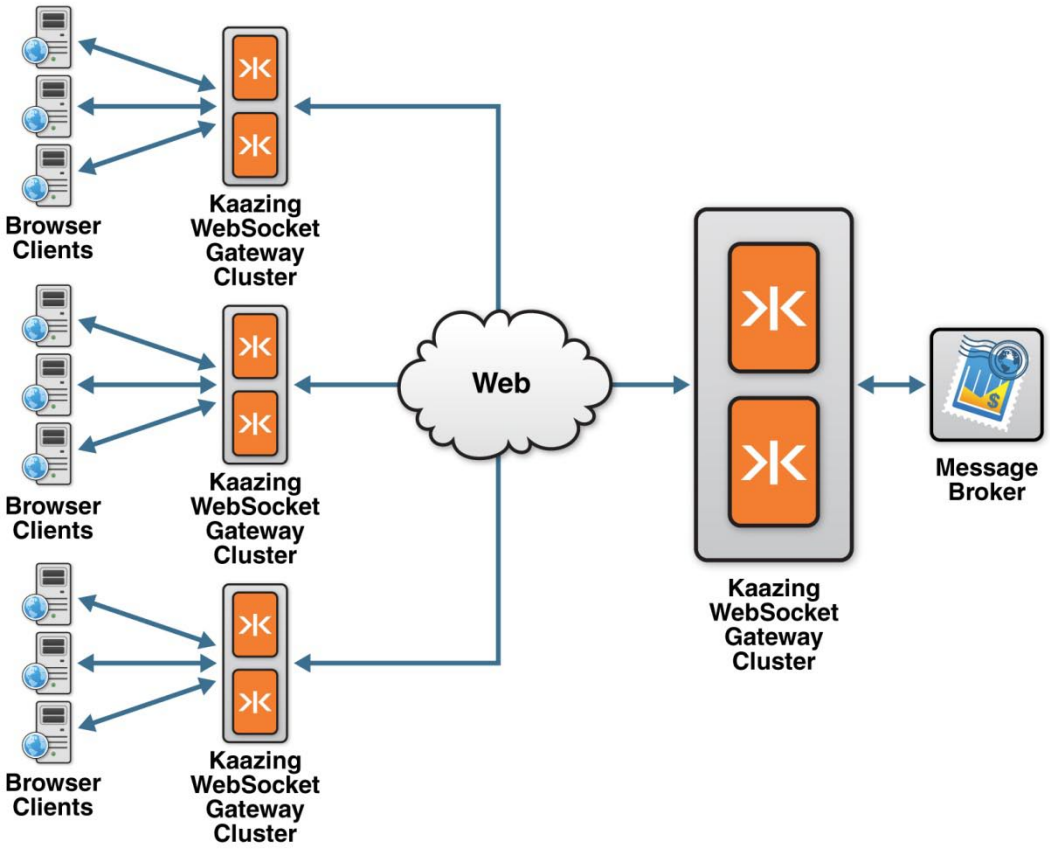

Figure 12—Fanout achieved by connection and subscription offloading

Stateful services share information between gateway instances to support failover. High availability for stateful services is achieved by configuring multiple gateway servers to be part of a cluster. Combining two or more gateways in a cluster creates a single entity from the client's point of view. In the clustered environment, essential state information is replicated for reliability so it can be recovered when needed (for example, when a hardware or software failover occurs).

Furthermore, stateful services in a cluster may share connections to a back-end service. Using gateway clusters enables the development of a hierarchy of gateways, using Kaazing WebSocket Gateways configured as Edge Gateways to be closer to the network edge in geographically dispersed locations. Using these clusters distributes the load, which scales architecturally, allowing far higher capacity deployments than other approaches. Figure 13 shows an example of a globally distributed network of Kaazing WebSocket Gateways.

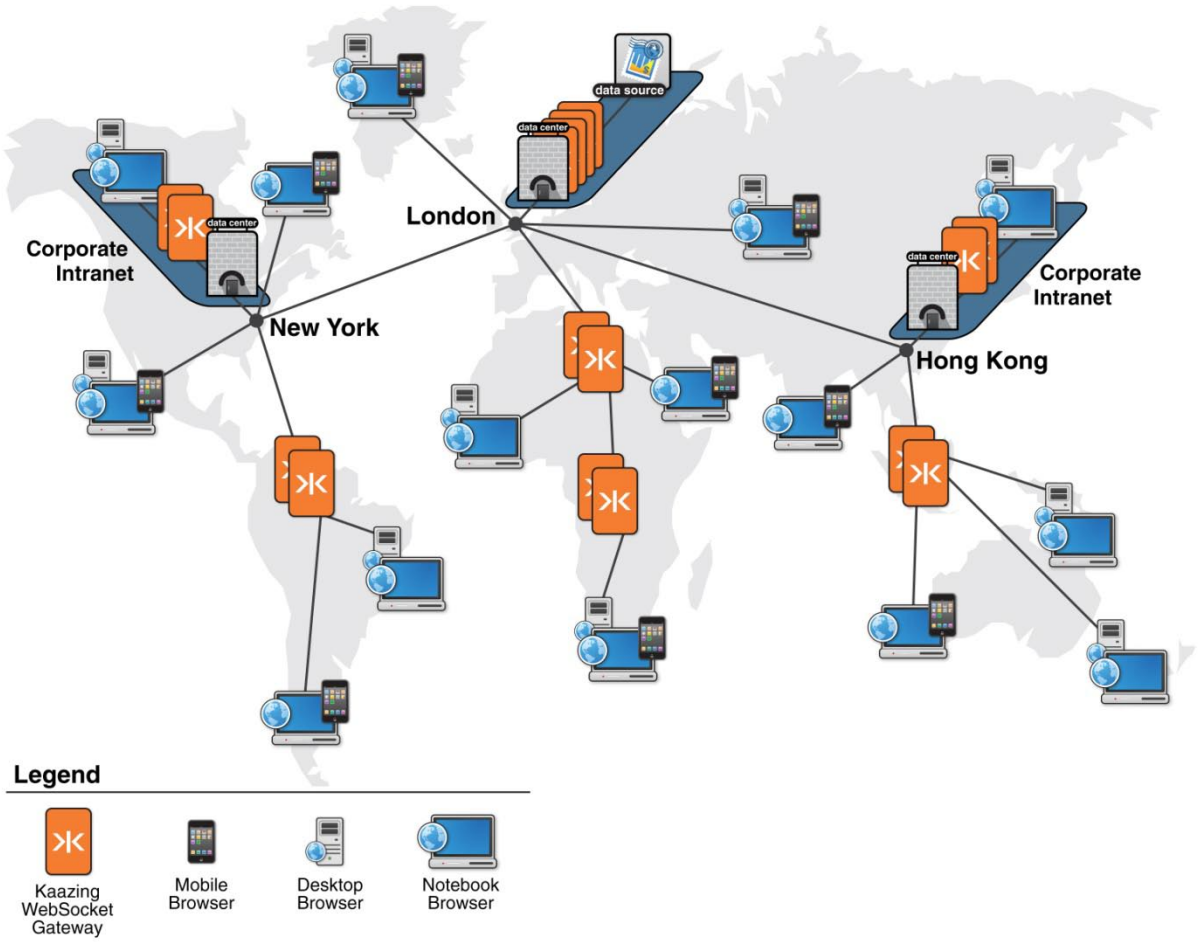

Figure 13—Global Kaazing WebSocket Gateway Distribution

KAAZING WEBSOCKET GATEWAY FEATURES

Kaazing WebSocket Gateway consists of the following components:

- One or more gateways (servers that act as a proxy to a back-end service) featuring Kaazing's WebSocket Acceleration. These handle connections to back-end data sources, and establish direct TCP connectivity between browsers and the back-end. They handle both native WebSocket connections and emulated WebSocket connections for older browsers.
- A set of enterprise-ready extensions for Kaazing WebSocket Gateway that offer additional features needed for enterprise deployment, such as protocol validation, encryption, support for shared credentials, and protocol adapters.
- A set of protocol-specific client libraries that developers use within their client-side applications, allowing applications written in JavaScript, Adobe Flex (Flash), Java/JavaFX, and Microsoft Silverlight to communicate directly with back-end servers in their native protocols. These libraries support emulated WebSocket for older browsers.

GATEWAY

Kaazing WebSocket Gateway's server component is an enterprise-grade HTML5 communication server that enables full-duplex TCP connectivity between browsers and back-end services. In case native support for WebSocket is not available in the browser, the socket connectivity between browsers and Kaazing WebSocket Gateway is emulated. The server component proxies communication to the remote server over raw TCP on behalf of clients.

Kaazing WebSocket Gateway is based on Staged Event-Driven Architecture (SEDA), and it leverages Java New I/O (NIO) for enhanced Java networking functionality. For example, instead of assigning a single thread per request, the server shares threads for optimal performance. By sharing the threads, Kaazing WebSocket Gateway server can achieve unparalleled levels of concurrent communication at extremely low message latency.

Unlike the HTTP request-response paradigm, WebSocket represents a connection-oriented architecture, which means that connections stay open between message transfers until the client explicitly closes them. Clearly, leaving a connection open has the potential to create additional overhead on the server, but this occurs only when the client anticipates server-initiated communication, such as subscribing to a stock ticker feed. The optimized WebSocket wire protocol lends itself to highly scalable WebSocket server implementations, where the transfer of each message consumes minimal resources, allowing many thousands of connections to be handled on a single server. Kaazing has successfully demonstrated one million connections to our Kaazing WebSocket Gateway to illustrate this point.

EXTENSIONS

To meet the needs of enterprise clients, Kaazing WebSocket Gateway includes a variety of extensions that enhance the basic server functionality. These include:

- **Protocol Validation** that improves perimeter security by verifying that bytes passing through Kaazing WebSocket Gateway on their way to the back-end server match the expected protocol wire format. This validation helps to prevent buffer overrun and other malicious attacks on the back-end server.
- **Dynamic Protocol Encryption** that allows Kaazing WebSocket Gateway to intercept communications with the back-end server and encrypt the WebSocket between the browser and Kaazing WebSocket Gateway, thereby satisfying the need for dynamic wire encryption without requiring the browser to implement wire encryption in JavaScript. For example, this feature allows the XmppClient client library to securely communicate with back-end XMPP servers, such as Google Talk, that require dynamic wire encryption after the initial handshake.
- **The Keyring Service** that can store client credentials in encrypted storage to expedite application startup while retaining a secure operating environment. Encrypted credentials sent to Kaazing WebSocket Gateway are automatically injected into the protocol before authenticating with the back-end system, which expedites application startup.
- **The Stomp-JMS Adapter Service** that lets Stomp client applications communicate directly with any JMS-compliant messaging broker.

CLIENT LIBRARIES

Kaazing WebSocket Gateway ships with a set of client libraries. Some of these enable emulated HTML5 communication on older browsers, while others make it easy to develop the client-side portion of specific protocols by simply including the library in the client application. Client-side libraries are available for different client technologies, such as JavaScript, Adobe Flex (Flash), Microsoft Silverlight, and Java/JavaFX, as shown in Figure 14.

***Note**: Kaazing is constantly enhancing and adding to the set of libraries available for Kaazing WebSocket Gateway. Refer to the Release Notes for the most recent list of client-side libraries and their supported client technologies.*

The following protocol-specific client libraries are available:

HTML5 Communication

- WebSocket
- ServerSentEvents
- ByteSocket

Commercial Protocols

- StompClient
- AmqpClient
- XmppClient
- IrcClient

Let's look at these libraries in more detail.

HTML5 Communication

WebSocket
The WebSocket client library can be used to open a bidirectional connection to communicate with a back-end service using text-based protocols, such as Jabber and IMAP. Once established, the connection remains open and both the client and the server can send information back and forth asynchronously. If the browser does not support WebSocket, the Kaazing WebSocket Gateway automatically falls back to emulated mode so that even older browsers can establish full-duplex communications.

ServerSentEvents
The ServerSentEvents client library allows clients to connect to any standards-compliant SSE stream. SSE standardizes and formalizes how a continuous stream of data can be sent from a server to a browser. If the browser does not support SSE natively, the library automatically falls back to emulated mode, letting modern browsers that do not include HTML5 capabilities stream events efficiently.

ByteSocket
The ByteSocket client library uses the same APIs as WebSocket. For HTML5-enabled browsers that support this function, clients and servers can establish a bidirectional connection that communicates directly to a back-end service using binary communication.

Commercial Protocols

StompClient
Streaming Text Orientated Messaging Protocol (Stomp) is a message transmission protocol. Implemented using the ByteSocket client library, Kaazing WebSocket Gateway's StompClient client library allows applications to send and receive messages to and from a Stomp-compliant server such as RabbitMQ, Apache ActiveMQ, or Tibco EMS. The StompClient client library allows developers to take advantage of many JMS features, like queuing, broadcasting, and publish and subscribe (pub/sub), in the same way that these JMS APIs can be used in Java, by using the Stomp-compliant server's JMS client library.

AmqpClient

Advanced Message Queuing Protocol (AMQP) is an open source message queue protocol designed for high-volume, low latency message bus communications. Implemented using the ByteSocket client library, Kaazing WebSocket Gateway's AmqpClient client library allows developers to send and receive messages to and from an AMQP server such as RabbitMQ, Apache Qpid, OpenMQ, Red Hat Enterprise MRG, ØMQ, and Zyre. Using the AmqpClient client libraries, developers can take advantage of the AMQP features, making the browser a first-class citizen in AMQP systems (similar to C, Java, Python, and other clients). This means that you can now run AMQP clients directly in a browser.

XmppClient

XMPP is a popular chat protocol used, for example, by Google Talk. Implemented using the WebSocket client library, the Kaazing WebSocket Gateway XmppClient client library allows developers to send and receive messages to and from an XMPP server and to take advantage of many XMPP features.

To support group chat, the XmppClient client library exposes the XmppRoom API. Because the implementation is layered on top of the WebSocket client library, Kaazing WebSocket Gateway provides accurate buddy lists by extending presence awareness all the way to the browser. This is a good example of the difference between "translated" client connections, where the web server would have to manage chatroom membership on behalf of clients, and true end-to-end connectivity provided by Kaazing WebSocket Gateway, in which each member of a chat is directly connected to the back-end server regardless of location or browser.

IrcClient

Internet Relay Chat (IRC) is a popular group chat system. Implemented using the ByteSocket client library, the IrcClient client library allows applications to send and receive messages to and from an Internet Relay Chat (IRC) server and to take advantage of many IRC features. The IrcClient client library uses Kaazing's ByteSocket client library, because IRC supports binary communication.

DEVELOPING APPLICATIONS WITH THE KAAZING CLIENT LIBRARIES

Kaazing WebSocket Gateway's documentation contains a set of how-to documents and tutorials that guide developers step-by-step through the creation of real-time applications that use HTML5 Communication, as well as those that build on Kaazing WebSocket Gateway's protocol clients that use WebSocket or ByteSocket internally. Additionally, the documentation library contains a tutorial on creating your own protocol client.

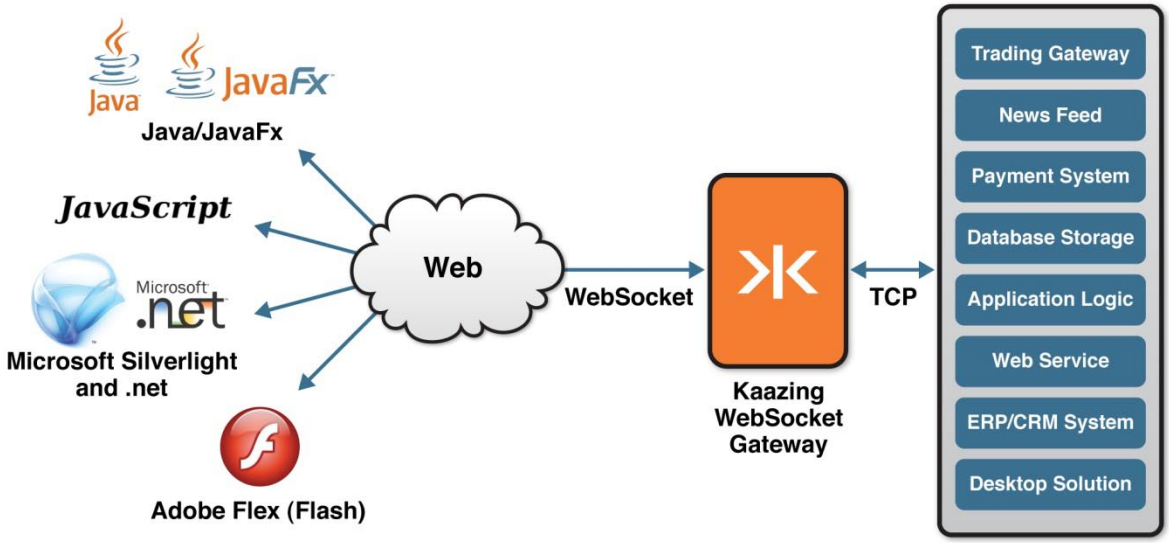

Figure 14—Kaazing WebSocket Gateway supports a wide range of client technologies

CLIENT TECHNOLOGY SUPPORT

Kaazing WebSocket Gateway works with a variety of Rich Internet Applications (RIAs) and client-side frameworks, ensuring that you can develop in the platform of your choice. These client-side technologies include JavaScript, Adobe Flex (Flash), Microsoft Silverlight, and Java/JavaFX, as shown in Figure 14. Refer to the *Release Notes* for information about which client technologies are supported for the different protocol client libraries.

BROWSER SUPPORT

Kaazing WebSocket Gateway is certified on the following major browsers, as shown in Figure 15:

- Apple Safari
- Google Chrome
- Microsoft Internet Explorer
- Mozilla Firefox
- Opera

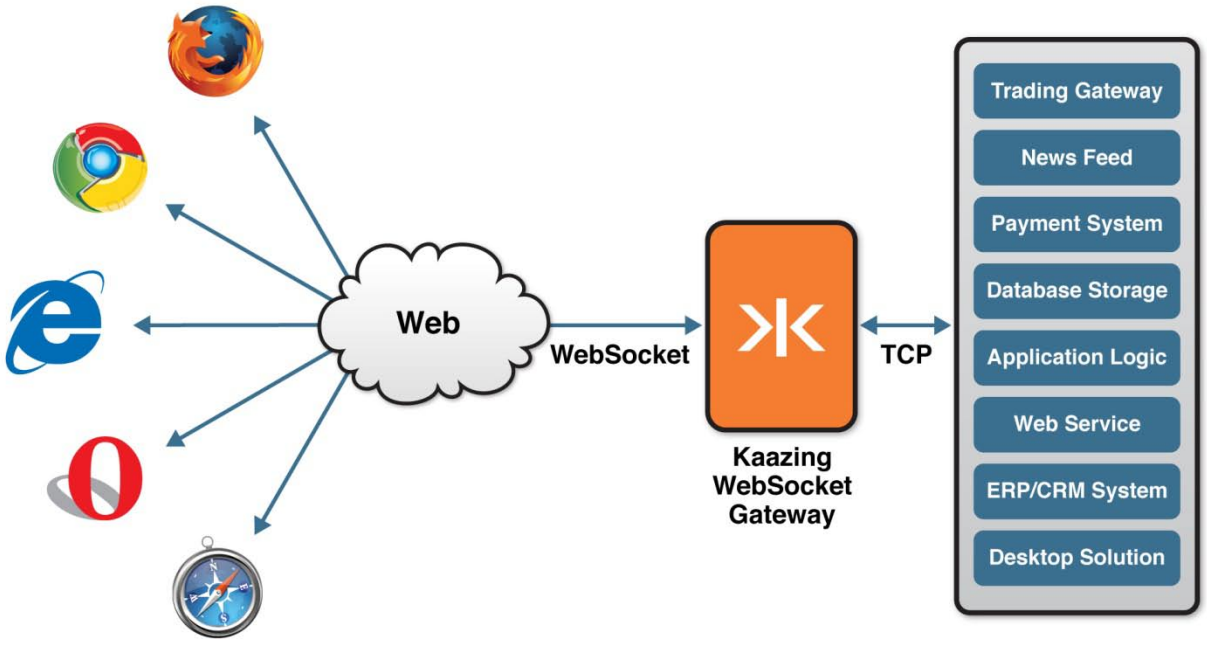

Figure 15—Kaazing WebSocket Gateway is certified on all major browsers

Refer to the *Release Notes* for information about the browser versions that are certified in the current release.

KAAZING SUPPORT OF HTML5 COMMUNICATION

HTML5 is an important shift in how the Internet communicates. It offers a variety of enhancements to browsers that change how applications are built and redefines what's possible online. From the early days of the specification, Kaazing has been at the center of the HTML5 WebSocket standard, working to ensure that scalable, full-duplex, real-time communication will be a reality on tomorrow's Web.

"Kaazing basically invented the protocol as it stands now"
– Ian Hickson, HTML5 Specification Editor, Google[2]

KAAZING AND HTML5 COMMUNICATION

The HTML5 specification defines both WebSocket and SSE. HTML5 WebSocket makes full-duplex and real-time communication a reality for browsers. As a result, this section of the HTML5 specification is subject to a great deal of attention from developers, analysts, and enterprises.

The first public draft of the HTML5 specification was published by the W3C in January 2008; however, the TCPConnection API and protocol were initially drafted even earlier than that, in 2007. With recent development and formalization, including significant input from Kaazing's founding team, the HTML5 Communication section has been solidified and is now nearly complete. This has prompted browser vendors to implement native support for the Communication section of the specification, despite the full HTML5 proposal remaining in draft form. Google Chrome and Safari already support WebSocket, and Firefox and Opera support is on the way.

Several portions of the HTML5 specification promise to fundamentally change how real-time web applications are created. These include WebSocket, SSE, Cross-Origin Resource Sharing, and Cross-Document Messaging.

WebSocket

With HTML5, the browser can set up a connection that both the browser and server can use to transmit data at the same time. What's more, because the connection doesn't need the HTTP metadata (such as HTTP status code and user-agent) it can be far more efficient than traditional HTTP requests; in fact, it behaves just like a raw TCP socket, and communication across WebSocket is dramatically faster and can deliver more information in fewer bytes.

The HTML5 WebSocket specification introduces the WebSocket interface, which defines a full-duplex communications channel that operates over a single socket. WebSocket:

- Traverses firewalls and routers seamlessly, because it looks just like HTTP to intermediate devices
- Allows authorized cross-domain communication
- Integrates with cookie-based authentication, since the HTTP connection that initiated the establishment of the WebSocket was a traditional HTTP request
- Integrates with existing HTTP load balancers

[2] Krijn Hoetmer's blog: http://krijnhoetmer.nl/irc-logs/whatwg/20100131

To establish a WebSocket connection, the client and server upgrade from the HTTP protocol to the WebSocket protocol during their initial handshake, as shown in Example 4.

Example 4—The WebSocket Handshake

```
GET /demo HTTP/1.1
Host: example.com
Connection: Upgrade
Sec-WebSocket-Key2: 12998 5 Y3 1  .P00
Sec-WebSocket-Protocol: sample
Upgrade: WebSocket
Sec-WebSocket-Key1: 4@1 46546xW%0l 1 5
Origin: http://example.com


HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Location: ws://example.com/demo
Sec-WebSocket-Protocol: sample
```

Once established, WebSocket data frames can be sent back and forth between the client and the server in full-duplex mode. To create a new WebSocket object, the client simply defines an object and the related listeners; at that point, it can post messages directly to the socket, as shown in Examples 5, 6, and 7.

Example 5—Using the WebSocket API to create a WebSocket object

```
var myWebSocket = new WebSocket
(“ws://www.websocket.org”);
```

Example 6—Adding WebSocket event handlers

```
myWebSocket.onopen = function(evt) { alert(“Connection open ...”); };

myWebSocket.onmessage = function(evt) { alert(“Received Message:  ”  +
evt.data);
};

myWebSocket.onclose = function(evt) { alert(“Connection closed.”); };
```

Example 7—Using the WebSocket API to send messages

```
myWebSocket.send(“Hello WebSocket!”);
myWebSocket.close();
```

Although the WebSocket protocol is ready to support a diverse set of clients, WebSocket cannot deliver raw binary data to JavaScript, because JavaScript does not support a byte type. Therefore, binary data is ignored if the client is JavaScript — but it can be delivered to other RIA clients that do support binary data.

WebSocket detects the presence of a proxy server and automatically sets up a tunnel to pass through the proxy. The tunnel is established by issuing an HTTP CONNECT statement to the proxy server, which requests for the proxy server to open a TCP/IP connection to a specific host and port. Once the tunnel is set up, communication can flow unimpeded through the proxy. Since HTTP/S works in a similar fashion, support for SSL is inherent. In addition to clear-text WebSocket connections, Kaazing WebSocket Gateway also supports Secure WebSocket connections (the 'wss' URI protocol).

Server-Sent Events

Another limitation of legacy browsers was their assumption that every web interaction began with a client’s request for content. This behavior worked fine for document-centric surfing of the kind envisioned by the web standards bodies, but it doesn’t take into account the full-duplex, real-time communications patterns that are common on the Web today.

By allowing the server and the client to both trigger events, applications can be truly real-time. Prior to HTML5, there was no way for a server to initiate communications (for example, when it had a message to send to the client) without the browser first asking for it. While developers have come up with a variety of ingenious hacks, from never-ending GETs to long-polling, to simulate server-initiated communication, all of these are plagued with performance and scalability issues.

HTML5 fixes this problem. SSE standardizes and formalizes how a continuous stream of data can be sent from a server to a browser. SSE is designed to enhance native, cross-browser streaming through the new EventSource JavaScript API that connects to a server URL to receive an event stream, as shown in Example 8.

Example 8—Using the EventSource API to create an EventSource

```
var eventSource = new EventSource("http://stock.example.com");
eventSource.onmessage = function(event) { alert(event.data); };
```

A connection to the server **stock.example.com** is established. When this server sends an event stream, a message event is dispatched to the **onmessage** handler. The server can then send the event shown in Example 9, where \n represents a newline character.

Example 9—Using the EventSource API to send a message

```
event: message\n
data: {'KZNG': 1.01}\n
\n
```

Next, the EventSource implementation in the browser dispatches `{'KZNG': 1.01}`. In this case, the stock price for KZNG is received by the client and displayed on the page.

Additionally, if the server includes the id header for an event, then the client adds a `Last-Event-ID` header when it reconnects, so that the event stream can resume without repeating or missing any events when the HTTP response completes, thus guaranteeing message delivery. The HTTP response can complete normally or else abruptly, if the underlying TCP connection is broken due to some network error. Furthermore, the server can control how long the client must wait before reconnecting by specifying an optional retry header as part of an event in the event stream.

Cross-Document Messaging

In the past, each document within a browser was isolated from other documents in order to prevent cross-site scripting attacks. The browser enforced a same-origin policy, preventing documents with different origins from affecting one other.

While isolating documents from each other provided good security, it also hobbled developers who wanted to send messages between documents for specific applications. To achieve this, Cross-Document Messaging provides a system that allows documents to securely communicate across different origins. As a result, the application can communicate with a server over several channels, separating the real-time, full-duplex message stream from other aspects of the web page or application.

The postMessage function, as defined in HTML5, allows communication between documents served by different origins. By calling postMessage on the target document's window, the sending application can pass the message and the expected origin of the destination to the target document, as shown in Example 10.

Example 10—Using the PostMessage API to send messages

```
document.getElementsByTagName('iframe')[0].contentWindow.postMessage
('KZNG', 'http://stock.example.com');
```

In this example, the message KZNG is sent to the document of the first iframe element, expecting the target origin to be `http://stock.example.com`. The receiving iframe, served by `http://stock.example.com`, must include an event listener to handle the incoming events, as shown in Example 11.

Example 11— Using the PostMessage API to handle incoming messages

```
function onmessage(event) {
  if (event.origin == 'http://www.example.com') {
    if (event.data == 'KZNG') {
      event.source.postMessage('KZNG:1.01', event.origin);
    }
  }
}
window.addEventListener('message', onmessage, false);
```

KAAZING WEBSOCKET GATEWAY IN THE ENTERPRISE

While HTML5, and WebSocket in particular, holds tremendous promise, customers demand high levels of reliability, security, and performance. They also require that applications be backwards compatible, integrating easily with established application protocols specific to their industries.

Kaazing WebSocket Gateway has been designed from the ground up with enterprise deployment in mind. Kaazing's founding team has years of experience building application software at some of the IT industry's largest software companies.

ENTERPRISE FEATURES

Kaazing WebSocket Gateway offers a robust, scalable architecture that can handle high message volumes, large numbers of concurrent users, low latency message delivery, and exceptionally high throughput. It works with a wide range of enterprise software protocols and development environments, and integrates easily with enterprise monitoring and management tools.

Let's take a closer look at how Kaazing WebSocket Gateway meets and exceeds enterprise IT requirements.

Reliability and resiliency

With Kaazing WebSocket Gateway, message delivery from back-end systems and services is guaranteed all the way from the server to the browser and vice-versa, in the same way that the HTML5 Specification guarantees TCP message delivery natively. In case of a broken request, Kaazing WebSocket Gateway automatically reconnects, guaranteeing message delivery.

Kaazing WebSocket Gateway also supports multicast so that a data source can be configured to send packets once to a multicast IP address (an IP address with the multicast address mask of 224.0.0.0). Multiple Kaazing WebSocket Gateways can be configured to receive the same multicast data.

This approach reduces complexity and provides increased scalability, because the data source does not have to be aware of all its clients and does not have to send each packet once to every Kaazing WebSocket Gateway in a high-availability cluster.

Standards compliance

Kaazing WebSocket Gateway is based on the HTML5 standard. Once browsers support full-duplex connectivity, per the HTML5 specification, there is no requirement to change any server or client code; applications automatically take advantage of the native implementations of HTML5 WebSocket and SSE with improved performance. Kaazing WebSocket Gateway's client libraries provide browser session storage and local storage as defined in the Storage section of the HTML5 specification.

What's more, while the WebSocket specification only supports text-based protocols, Kaazing WebSocket Gateway also supports binary protocols, enabling raw TCP communication between client and server.

Scalability and performance

Kaazing WebSocket Gateway scales easily to enormous amounts of concurrent users. Kaazing WebSocket Gateway runs separately from application logic hosted in an application server, which mitigates competition for resources. It also supports broadcast notification of data sent from a back-end service to all connected clients, transmitting UDP messages to browsers via Server-Sent Events. In addition, Kaazing WebSocket Gateway's ability to offload connection overhead and to create a hierarchy of gateways to distribute load scales architecturally, allowing far higher capacity deployments than other approaches.

By using HTML5 WebSocket, data between the client and back-end server is no longer encumbered by the HTTP header overhead of other approaches. For the browser, this means that rather than contacting back-end servers through custom servlet intermediaries or complex application server logic; it can now send binary data packages and receive an immediate response.

Backwards compatibility and legacy support

Kaazing WebSocket Gateway is certified on all major browsers. Refer to the *Release Notes* for information about the browser versions that are certified in this release. Even for browsers that don't support HTML5, Kaazing WebSocket Gateway features WebSocket emulation that makes WebSocket available in all browsers. This emulation works in a pure JavaScript environment without plugins, and features Kaazing's unique Opportunistic Optimization™ technology that ensures the best possible connection environment, whether or not clients and intermediate proxy servers support the latest protocols.

For example, if Kaazing WebSocket Gateway detects the presence of the Flash plugin, client libraries can take advantage of the single (Flash) TCP socket connection and if a direct connection is not possible (for example, if communication must flow through a firewall or an HTTP proxy server), then client libraries can still take advantage of the Flash runtime, minimizing the client's memory profile.

If intermediate proxy servers lie between Kaazing WebSocket Gateway and a client, then a highly optimized encrypted streaming connection is used and the proxy-server-aware gateway automatically redirects the HTTP request so that it uses an encrypted HTTP (HTTPS) connection to make the proxy server agnostic to the streaming nature of the downstream HTTP traffic. In a production environment, one normally must anticipate that HTTPS streaming can be used as a fall back in a worst-case scenario. However, for this to work, the Gateway has to be configured with a suitable certificate for TLS encryption. Without that (this should be considered a configuration error), Kaazing will fall back to an advanced non-streaming implementation.

Kaazing's WebSocket emulation is highly optimized and will easily outperform most legacy Comet solutions, because it can guarantee minimal latency by falling back to HTTPS streaming, thanks to Kaazing's underlying support for cross-origin HTTP and HTTPS requests. Note that the different fall-back scenarios are only used in emulated mode. One major benefit of using Kaazing WebSocket Gateway and its WebSocket emulation is that you can code applications against the HTML5 WebSocket standard, today, and these applications will work in all browsers.

Rapid application development and deployment

If you're trying to prototype, test, and deploy applications quickly, Kaazing WebSocket Gateway provides a set of protocol-specific client libraries for different client technologies. On the other hand, if you're porting an entirely new protocol to a web-based environment, we offer detailed information on how to create new protocol-specific libraries that build on top of the WebSocket and ByteSocket libraries. By starting with these libraries, you ensure a fast, web-based protocol that's backwards compatible—rather than having to develop your own web-side protocol that deals with legacy clients, disconnection, and guaranteed delivery.

Coexistence with other web servers

While Kaazing WebSocket Gateway is focused on fast, real-time, bidirectional communications, it can also serve static HTML content, making it easy to build the front-end required for client welcoming and authentication. Kaazing WebSocket Gateway can also be used in conjunction with a web server (such as the Apache Server) or a web application server. Refer to the *Administrator's Guide* for information about configuring different web servers to integrate with Kaazing WebSocket Gateway.

Security

Kaazing WebSocket Gateway has built-in protocol awareness, verifying that bytes flowing through the gateway match the expected protocol wire format. This check helps to prevent buffer overrun attacks or other malicious attacks on the back-end server.

To encrypt transmitted data, Kaazing WebSocket Gateway supports wire encryption using HTTPS, W3C Cross-Origin Resource Sharing, and HTTP-based authentication and authorization. It also integrates with JAAS, supporting pluggable authentication modules. Client libraries provide encrypted storage to remember back-end system credentials. Encrypted credentials sent to Kaazing WebSocket Gateway are automatically injected into the protocol before authenticating with the back-end system, eliminating the time and risk of a long, multi-password authentication sequence without compromising credential storage.

Kaazing takes security very seriously, and Kaazing WebSocket Gateway was purpose-built for environments with stringent security needs. Refer to Kaazing's *Security Overview* for more information about how security is implemented. This document provides an in-depth look at how Kaazing security works under the covers by taking a look at a few sample scenarios that leverage the following aspects of security:

- Credential Caching Strategies
- PlaintextKeyring API
- EncryptedKeyring API
- Wire Traffic Encryption
- HTTP Authentication and Authorization
- Encrypted Session Cookies
- Cookies and Domain Limitations
- Single Sign-On
- Kaazing Keyring Service
- Kaazing Protocol Validation and Credential Injection
- Cross-Origin Resource Sharing

Manageability

Kaazing WebSocket Gateway exposes a set of Java Management eXtensions (JMX) MBeans that allow administrators to manage and monitor the internal state of the gateway. In addition, the Kaazing Stomp-JMS Adapter enables Stomp client applications to communicate directly with any JMS-compliant messaging broker. This direct communication enables management tools compatible with these protocols to recognize Kaazing WebSocket Gateway and connected Kaazing WebSocket Gateway clients as part of their management topography, letting IT teams use existing management tools to oversee both LAN-based and browser-attached users. Kaazing WebSocket Gateway also offers a variety of management interfaces and logs information to standard logging collectors.

Virtual Private Cloud Connection

Many of the examples we've looked at relate to browser-to-server communication. It's important to remember, however, that WebSocket-based communication works equally well for machine-to-machine communication. WebSocket offer the performance characteristics of a native TCP connection, with the compatibility and end-to-end traversal capabilities of conventional web protocols. Hence, WebSocket is well suited for emerging applications such as cloud-to-cloud connectivity and multi-tiered message distribution. To this end, Kaazing WebSocket Gateway can be configured to create a virtual private connection for machine-to-machine communication.

SUMMARY

Until now, it's been impossible to achieve true bidirectional communication between clients and servers on the Internet. Most approaches have relied on protocol trickery and inefficient communications models, which has hampered advances in rich client applications.

HTML5 promises to change this, adding critical functions such as WebSocket and SSE to web developers' toolboxes. To deploy and deliver fast, scalable applications — and to support even older browsers that haven't yet implemented these standards — Kaazing has created Kaazing WebSocket Gateway.

Kaazing WebSocket Gateway consists of a server-side gateway and client libraries that make it easy to build JavaScript/Ajax, Java/JavaFX, Adobe Flex (Flash) and Microsoft Silverlight-based applications that can communicate bidirectionally with back-end servers. Kaazing WebSocket Gateway's unique WebSocket Acceleration technology, underlying SEDA architecture, and Opportunistic Optimization offer significantly lower latency, higher capacity, and improved robustness when compared with other solutions.

Kaazing WebSocket Gateway lets browsers open a socket connection to any TCP-based back-end service (such as JMS, JMX, IMAP, or Jabber servers). This radically simplifies the convoluted Java EE architectures of the past, reducing the number of machines needed to deliver content quickly, speeding up development cycles, and slashing IT operations and support costs. Finally, the browser is a first-class citizen of network communications, a benefit that has long been enjoyed only by desktop applications.

To learn more about Kaazing, Kaazing WebSocket Gateway, or the ways that HTML5 will transform the face of the web, visit us at **www.kaazing.com**.

NEXT STEPS

The Kaazing WebSocket Gateway documentation library contains a *Getting Started* that helps you start the demo bundle quickly, as well as an in-depth *Administrator's Guide*, which contains all the information you'll need to quickly deploy and configure Kaazing WebSocket Gateway. It includes information on the following topics:

- Configuring Kaazing WebSocket Gateway
- Securing Kaazing WebSocket Gateway
- Monitoring Kaazing WebSocket Gateway
- Configuring Kaazing WebSocket Gateway to use multicast
- Configuring a web server to integrate with Kaazing WebSocket Gateway
- Troubleshooting

