Elko Technology Overview

1-February-2016

Elko provides a suite of servers for hosting highly scalable, sessionful internet applications, along with a set of common platform components upon which these servers are all based. Although the Elko server suite provides a complete platform for most kinds of server applications, the underlying components can readily be used for the creation of additional types of servers should they be needed.

The Elko framework is targeted at stateful applications, in contrast to traditional web servers, which are (nominally) stateless. Most real-world internet applications, and certainly those that attempt to deliver a real-time interactive experience, entail significant short term and long term application state that must be maintained somewhere. Web servers attain scalability by virtue of their statelessness, which enables capacity expansion by simple server replication, but this scale comes at the expense of considerable added complexity and inefficiency once the need to maintain application state is taken into account. Moreover, much of this complexity is often inflicted directly on application developers in ways that slow development and impede software maintainability. Elko servers, in contrast, maintain application state natively in an immediate and developer friendly way, and achieve massive scalability by other means.

An important class of real-time internet applications for which the Elko framework is particularly well suited are networked games. Consequently, an important part of the technology is an additional set of service frameworks and object classes that build on top of the basic server infrastructure specifically to support real-time, geo-enabled multiplayer games to clients on a variety of different client platforms, most notably mobile devices, in addition to traditional web, PC, and console environments. This support includes client-side libraries for a number of key client platforms; these libraries provide integrated client-server communications, message dispatch, and object life-cycle management.

The Elko Server Suite

The Elko server suite consists of a family of different types of servers that cooperatively form a highly scalable application hosting system for real-time, stateful applications. The current server suite consists of seven different kinds of servers:

Context server — where applications run
Director — directs users to contexts and load balances the pool of context servers
Presence server — tracks user presence and manages social graphs
Workshop — provides a place for arbitrary cross-server services to run
Broker — central admin and configuration management for all the other servers
Gatekeeper — adds support for external user authentication and login handling
Repository server — provides persistent object storage services

Context Server

The *context server* is the heart of the Elko server suite. It is the place where application-specific classes are loaded and run to realize the particular objects and behavior that distinguish one game or application from another. The other types of servers play important supporting roles in making the whole system work effectively, but, aside from a few specialized use cases, an application developer does not need to be concerned with the details of anything but the context server.

As its name suggests, the context server manages entities we call *contexts*. A context is a rendezvous and coordination point for communications among multiple parties. What a context represents in a particular application depends on the application, but examples might include a chat room in a multi-user chat application, an auction in a real-time auction application, a poker table in a real-time gambling game, or a small neighborhood in a virtual world game. A context provides a scope for object addressing and message routing — a common frame of reference for interaction. The context abstraction is concerned with the things that are visible in common to a set of concurrently connected users.

The context server manages network connections with its clients, allowing each client to have a real-time, full-duplex, asynchronous, object-to-object message pipe between application code running in the client device and application objects hosted within the context server itself. The server handles connection setup and teardown, serialization and deserialization of messages, message routing and delivery, and automatic dispatch of each incoming message to the application objects and methods on the server that should handle it. The server can also handle the delivery of messages from application code on the server to individual clients, as well as the fanout of messages to arbitrary sets of relevant clients (such as, for example, all the users in a particular context).

The context server also includes an optional quadtree based object lookup system, which allows users and other objects to be identified by geospatial position information. This enables the server to, for example, fan a message to all the users whose devices report GPS coordinates within some radius of another user or who are within some geographic region of interest to the application.

In addition to handling the serialization and deserialization of messages, the context server also handles the serialization and deserialization of objects as they are moved into and out of persistent storage. The server by default automatically manages the persistence of contexts and the objects they contain as they are activated and deactivated, and of users and the objects *they* hold as they come and go from the server. The persistence mechanism also allows for explicit checkpointing of object state to persistent storage in cases where application requirements demand that critical object state changes be managed transactionally.

Context servers are very efficient and highly scalable. Exactly how scalable depends, of course, on the backend computational demands of the particular game or application as well as the typical kinds of message fanouts expected (that is, the average number of outgoing messages to different clients triggered by a single incoming message from one client – this can range from one to several thousands depending on the nature of the application). However, for typical applications, both scale tests and operational experience over many years indicate that a single context server can generally support 50,000 to 200,000 or more concurrently connected users on a typical enterprise class server machine (such as a single Amazon EC2 "large" instance).

Director

The simplest possible Elko configuration consists of a single context server. However, if the number of concurrent users that must be supported exceeds the capacity of a single machine, more context servers may be added for scale, at which point one or more *directors* are required. The director is a specialized server that acts as a traffic cop and load balancer. A director keeps track of the set of running context servers, as well as what contexts are active on each and which users are in which contexts. When a user client wishes to enter a context, it talks to the director to find out which context server to go to. Since the director knows where each context is running, it can direct the client accordingly. If a given context is not active on any context server when a user seeks to enter it, the director can choose a suitable context server to host it and direct that server to activate the context before sending the user there. By employing a director, any context can be run on any context server in a way that is completely transparent to the user. Furthermore, loads are automatically balanced across the constellation of context servers without any operator intervention or manual configuration management being required. The director also manages failover in the unlikely event that a particular context server crashes.

Since the job that it does is ultimately quite simple, and the client interactions with it are quite brief, a single director can manage a very large number of context servers and an extremely large number of users. Even so, directors can be replicated for additional scale if the job requires it, with traffic from clients to the multiple directors in turn distributed via conventional data center load balancing routers. In practice, scaling for directors is rarely an issue, but it is common to have at least two directors to provide redundancy for reliability and fault tolerance.

Presence Server

The *presence server* enables a context-hosted application to integrate with social graph data, including both external graphs managed outside the Elko server cluster (such as a user's Facebook friends) and internal graphs maintained within the presence server itself according to application-provided logic (for example, a competitive game might choose to keep track of players' in-game rivalries). The presence server monitors the coming and going of users to and from the various context servers, so it knows at all times where each currently connected user is. Application code (running in the context servers) can

subscribe to notifications about these users' comings and goings as viewed through the lens of other users' social graphs. For example, a context serving an instance of the competitive game alluded to above could arrange to be notified whenever a rival of one of its users arrived in some other context, in order to facilitate setting up a real-time challenge match. In a similar vein, a chat application could use the presence server to monitor the real-time presence of each of its users' Facebook friends.

The presence server can also act as a message relay between users across context boundaries, even if these users are on different context servers. Presence monitoring and inter-context messaging is possible even across application boundaries, enabling, for example, cross game notifications. Such notifications could be used by a game publisher to engage in real-time cross promotions among its various game titles, or even between its titles and those of other publishers with whom it has arranged some kind reciprocal marketing agreement.

Workshop

Some games and other applications require access to services that are not specific to any particular context but are global to the entire application (or collection of related applications). These services, whether considered part of the application or independent of it, may nevertheless need to make use of real-time state or have autonomous processes that need to run separately from any particular client. The *workshop* is a server that provides a place for these kinds of services to be hosted. It also provides a registry that allows a service to advertise its availability and allows clients (in this case, application or game code running in the various context servers) to locate and access the services they need.

The actual services themselves may be application specific functions that are provided and installed in the workshop by the application developer, or they may be instances of services that are provided by the Elko framework itself. The Elko distribution provides a number of generally useful workshop services as part of the standard environment available to all applications. Among these are a secure transactional bank for virtual currencies and a wrapper for asynchronous access to arbitrary external webservices.

Broker

If configured individually, managing a system composed of all of these disparate servers could entail a lot of administrative complexity. The *broker* simplifies that, by providing a one-stop centralized administrative control point for managing a large cluster of servers. The broker enables other servers to be added and removed dynamically, and isolates the individual servers from concern with overall configuration details or with order-of-startup issues. It also provides a centralized facility for server monitoring and metrics collection.

In conjunction with the broker, the server suite also includes a web-based management console, allowing a server cluster to be monitored and configured from a simple, easy to use web interface.

Other servers

In addition to the above described components, the Elko server suite includes a few other, more minor pieces.

While context servers and directors themselves can do basic user authentication, more sophisticated authentication and user account schemes may require the addition of one more *gatekeepers*, which can integrate with external account management, identity, and authentication systems. A gatekeeper allows the developer to insert their own identity management and user authentication mechanism in between the user's client and the other servers that the user client interacts with directly (i.e., the context server and director). This enables a Elko server cluster to interoperate with proprietary or legacy identification and authentication schemes.

The context server (and a few of the other servers) saves the persistent state of objects in an external store that we call the *repository*. The repository interface is abstract, enabling different implementations to be supported for different purposes. The standard implementations include a simple, flat-file based store, useful for configuration files and other such basic uses, as well as a full-featured object store based on MongoDB, a widely used open source NoSQL database. Other types of repository, such as something based on a relational database such as MySQL or Oracle, can readily be implemented. The server suite also includes a *repository server*, which is a repository implementation that wraps the standard repository interface in a server of its own. By plugging your own repository implementation beneath this, it is possible to transform what is on otherwise non-sharable storage medium into a sharable resource (for example, MongoDB or MySQL are intrinsically shareable, and so don't require this treatment, whereas the flat-file store is not intrinsically shareable, but can be made shareable by wrapping a repository server around it).

Since all these servers are constructed using a common set of core building block classes, it is also relatively easy and fast to use Elko to generate new species of servers as requirements emerge. Such needs emerge rarely, in our experience, but the framework can readily accommodate these cases when they do.

Nuts and Bolts

JSON Messaging

Communications between clients and the context server, as well as among the various different servers on the backend, is via *JSON messaging*.

JSON messaging is a set of conventions for encoding object-to-object messages in JSON. A JSON message is simply a JSON object of the form:

```
{ to:targetRef, op:verb, params... }
```

The property `to` designates the message target, i.e., the object to which the message is addressed. Its value is a *ref*, a string that uniquely names the target object in the scope of the messaging system on the arriving end of the communication.

The property `op` is the message verb, i.e., the operation code or method selector, a string that indicates to the message target which operation is to be performed upon receipt of the message.

Any number of other properties may also be included, and serve as named parameters. Their number, names, and meanings vary depending on the specific message being sent. They may be of any data type supported by JSON, as long as there is consistent mutual understanding between the sender and receiver of the proper content for the message. The order of the parameters is not significant, though in documentation we conventionally write the `to` and `op` parameters first for clarity of presentation. (Note that to improve documentation legibility, we also use a slightly augmented JSON syntax, where we skip the quotation marks around the property names. This notation is also understood by the server, as it considerably mitigates the drudgery of manual composition of JSON expressions during debugging. The server does, however, conform to the strict RFC 7159 JSON standard in the things it outputs unless explicitly configured not to.)

For example, the message:

```
{ to:"u-47-3699102", op:"say", utterance:"Hi Fred!" }
```

might be a chat message directed to another user.

JSON messages may be transmitted over any reasonable communications medium. The current implementation supports raw TCP, HTTP, WebSockets, and ØMQ, as well as the SSL variants of these (HTTPS, etc). We also support a protocol of our own that we've named Resumable TCP (RTCP), which adds a simple session layer on top of TCP to support reliable communications over unreliable wireless networks, specifically for mobile devices.

A message connection is always a bidirectional, multiplexed, machine-to-machine (or process-to-process) message pipe. Two communicating machines, whether they are a client and a server or a server and another server, typically maintain a single connection between them, over which all message traffic on behalf of objects on either side is carried. A given connection is kept alive for the duration of the interaction between the two machines, until one side or the other decides to terminate it. This communications session is stateful, in that either party is permitted, indeed expected, to maintain

continuity of state between successive messages. This is directly in contrast to stateless, sessionless protocols such as HTTP.

Though a connection is bidirectional, messaging over the connection is unidirectional and asynchronous. That is, as soon as a running application passes a message to the messaging system, it is able to continue on its own without waiting for reply or confirmation. When a message is part of a request-reply protocol, the reply must be treated as an explicit, separate return message by the two parties. Any such protocol is part of the design of the particular application in question and is not the business of the messaging system itself. Any given message may result in a reply from its recipient, no reply, or possibly even many replies, depending on the design of the application and its protocols.

Messages on a connection are ordered, in the sense that a sequence of messages transmitted over a given connection from machine A to machine B will be processed at machine B in the same order they were transmitted by machine A. However, this ordering is strictly on a per-connection basis. If there is more than one connection between two machines (not normal but not forbidden either), there are no guarantees about the relative order of messages sent over different connections.

Since HTTP is stateless and sessionless, an additional transport protocol is used when the message transport medium is HTTP. This protocol uses special URLs and additional JSON encoding within the HTTP request and reply bodies to efficiently synthesize a sessionful, symmetric, bidirectional, asynchronous message channel out of an ongoing series of transient, asymmetric, synchronous, RPC-style HTTP requests. The details of this transport protocol are documented elsewhere.

Object State Representation

For purposes of transmission in messages and writing to persistent storage, the state of an object is serialized into JSON following a few simple representational conventions. The conventional form of an object is:

```
{ type:typeTag, ref:refString, properties... }
```

The property `type` encodes the data type of the object. This is a simple tag string that is mapped to the actual object class internally. On the server, a mapping table, itself stored as a persistent object in the server's object repository, maps the type tag to the fully qualified Java class name of the class whose constructor will decode the JSON serialized representation into a live Java object in the server's memory. (We use a tag string rather than using the class name directly in order to decouple the JSON form from server-specific implementation details. It also reduces the size of the representation, cutting bandwidth and storage requirements.) Depending on circumstances, the `type` property is not always required, since the types of pure data objects embedded within other objects can frequently be determined by the context in which they are used. For example, say you had a polygon object, one of whose properties is an array of points, e.g.:

```
{ type:"poly", points:[point, point, ...], whatever... }
```

you might simply represent an individual point as:

```
{ x:xVal, y:yVal }
```

rather than, say:

```
{ type:"point", x:xVal, y:yVal }
```

since at the place in the code where it needs to deserialize the array of points, the code already knows that it's looking for an array of points, and redundantly tagging each of them would just be wasteful.

The property `ref` is the object's ref, the unique identifier that designates that specific object instance. The `ref` property is only present when the object in question is referenceable, i.e., if it can be pointed to by something else, most notably as the designated target of a message send. Pure data objects that are simply structured property values inside other objects are typically not referenceable in this way, and so their JSON representations would omit this property.

The remaining properties depend entirely on the object type. It is the responsibility of the code that serializes and deserializes a given type to interpret or generate these properties as appropriate.

Note that this JSON representation is a serialization scheme used for communicating an object's state from one place to another. What it becomes when interpreted by the receiving entity is entirely undefined here. We loosely talk about storing a serialized object in the repository, but how the repository actually represents things internally is its own business. It is not required, or even really expected, that the repository will be storing and retrieving actual literal JSON strings.

