# Clustering

# Session clustering

TomEE fully relies on Tomcat clustering: Tomcat Clustering.

The configuration is mainly in `conf/server.xml` and since TomEE 7 CDI `@SessionScoped` is transparently clustered through the session.

# Hazelcast as session provider

Hazelcast did a post on this topic on Session Clustering With TomEE.

Tomitribe also demonstrated you can distributed `@Stateful` beans easily relying on hazelcast: Hazelcast TomEE PoC.

# Load balancing

TomEE being a HTTP server all HTTP load balancer such as HTTPd (a.k.a. Apache2), ngnix, F5 etc... will work.

More documentation on HTTPd link can be found on Tomcat website.

# EJBd

If you use the EJBd protocol (`@Remote` EJB proprietary protocol of TomEE) you can get cluster features on the client part.

## Multicast

Multicast is the preferred way to broadcast the heartbeat on the network. The simple technique of broadcasting a non-changing service URI on the network has specific advantages to multicast. The URI itself is essentially stateless and there is no "i'm alive" URI or an "i'm dead" URI.

In this way the issues with UDP being unordered and unreliable melt away as state is no longer a concern and packet sizes are always small. Complicated libraries that ride atop UDP and attempt to offer reliability (retransmission) and ordering on UDP can be avoided. As well the advantages UDP has over TCP are retained as there are no java layers attempting to force UDP communication to be more TCP-like. The simple design means UDP/Multicast is only used for discovery and from there on out critical information is transmitted over TCP/IP which is obviously going to do a better job at ensuring reliability and ordering.

### Server Configuration

When you boot the server there should be a conf/multicast.properties file containing:

```
server      = org.apache.openejb.server.discovery.MulticastDiscoveryAgent
bind        = 239.255.2.3
port        = 6142
disabled    = true
group       = default
```

Just need to enable that by setting disabled=false. All of the above settings except server can be changed. The port and bind must be valid for general multicast/udp network communication.

The group setting can be changed to further group servers that may use the same multicast channel. As shown below the client also has a group setting which can be used to select an appropriate server from the multicast channel.

| | |
|---|---|
| **IMPORTANT** | for multicast to work you need to have ejbd activated as a normal service. This can be done setting in `conf/system.properties` the entry: `openejb.service.manager.class = org.apache.openejb.server.SimpleServiceManager`. |

**Multicast Client**

The multicast functionality is not just for servers to find each other in a cluster, it can also be used for EJB clients to discover a server. A special multicast:// URL can be used in the InitialContext properties to signify that multicast should be used to seed the connection process. Such as:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.RemoteInitialContextFactory");
p.put(Context.PROVIDER_URL, "multicast://239.255.2.3:6142?group=default");
InitialContext remoteContext = new InitialContext(p);
```

The URL has optional query parameters such as schemes and group and timeout which allow you to zero in on a particular type of service of a particular cluster group as well as set how long you are willing to wait in the discovery process till finally giving up. The first matching service that it sees "flowing" around on the UDP stream is the one it picks and sticks to for that and subsequent requests, ensuring UDP is only used when there are no other servers to talk to.

Note that EJB clients do not need to use multicast to find a server. If the client knows the URL of a server in the cluster, it may use it and connect directly to that server, at which point that server will share the full list of its peers.

**Multicast Servers with TCP Clients**

Note that clients do not need to use multicast to communicate with servers. Servers can use multicast to discover each other, but clients are still free to connect to servers in the network using the server's TCP address.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.RemoteInitialContextFactory");
p.put(Context.PROVIDER_URL, "ejbd://192.168.1.30:4201");
InitialContext remoteContext = new InitialContext(p);
When the client connects, the server will send the URLs of all the servers in the
group and failover will take place normally.
```

## Multipulse

MultiPulse is an alternative multicast lookup that does not use a regular heartbeat. Instead, servers listen for a multicast request packet (a pulse) to which a response is then sent. Multicast network traffic is effectively reduced to an absolute minimum.

MultiPulse is only useful in network scenarios where both client and server can be configured to send multicast UDP packets.

### Server Configuration

After you boot the server for the first time the default configuration will create the file conf/conf.d/multipulse.properties containing:

```
server      = org.apache.openejb.server.discovery.MulticastPulseAgent
bind        = 239.255.2.3
port        = 6142
disabled    = true
group       = default
```

You just need to enable the agent by setting disabled = false. It is advisable to disable multicast in the multicast.properties file, or at least to use a different bind address or port should you wish to use both.

All of the above settings except server can be modified as required. The port and bind must be valid for general multicast/udp network communication.

The group setting can be changed to further group/cluster servers that may use the same multicast channel. As shown below the client also has an optional group setting which can be used to select an appropriate server cluster from the multicast channel (See MultiPulse Client).

The next step is to ensure that the advertised services are configured for discovery. Edit the ejbd.properties file (and any other enabled services such as http, etc.) and ensure that the discovery option is set to a value that remote clients will be able to resolve.

```
server      = org.apache.openejb.server.ejbd.EjbServer
bind        = 0.0.0.0
port        = 4201
disabled    = false
threads     = 20
discovery   = ejb:ejbd://{bind}:{port}
```

| NOTE | If either 0.0.0.0 (IPv4) or [::] (IPv6) wildcard bind addresses are used then the server will actually broadcast all of it's known public hosts to clients. Clients will then cycle though and attempt to connect to the provided hosts until successful. |
|---|---|

If localhost is used then only clients on the same physical machine will actually 'see' the server response.

#### MultiPulse Client

The multipulse functionality is not just for servers to find each other in a cluster, it can also be used for EJB clients to discover a server. A special multipulse:// URL can be used in the InitialContext properties to signify that multipulse should be used to seed the connection process. Such as:

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.RemoteInitialContextFactory");
p.put(Context.PROVIDER_URL, "multipulse://239.255.2.3:6142?group=default&timeout=250"
);
InitialContext remoteContext = new InitialContext(p);
```

The URL has optional query parameters such as schemes and group and timeout which allow you to zero in on a particular type of service of a particular cluster group as well as set how long you are willing to wait in the discovery process till finally giving up. The first matching service that it sees "flowing" around on the UDP stream is the one it picks and sticks to for that and subsequent requests, ensuring UDP is only used when there are no other servers to talk to.

Note that EJB clients do not need to use multipulse to find a server. If the client knows the URL of a server in the cluster, it may use it and connect directly to that server, at which point that server will share the full list of its peers.

Multicast Servers with TCP Clients

Note that clients do not need to use multipulse to communicate with servers. Servers can use multicast to discover each other, but clients are still free to connect to servers in the network using the server's TCP address.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.RemoteInitialContextFactory");
p.put(Context.PROVIDER_URL, "ejbd://192.168.1.30:4201");
InitialContext remoteContext = new InitialContext(p);
```

When the client connects, the server will send the URLs of all the servers in the group and failover will take place normally.

## Multipoint

As TCP has no real broadcast functionality to speak of, communication of who is in the network is achieved by each server having a physical connection to each other server in the network.

To join the network, the server must be configured to know the address of at least one server in the network and connect to it. When it does both servers will exchange the full list of all the other servers each knows about. Each server will then connect to any new servers they've just learned about and repeat the processes with those new servers. The end result is that everyone has a direct connection to everyone 100% of the time, hence the made-up term "multipoint" to describe this situation of each server having multiple point-to-point connections which create a fully connected graph.

On the client side things are similar. It needs to know the address of at least one server in the network and be able to connect to it. When it does it will get the full (and dynamically maintained) list of every server in the network. The client doesn't connect to each of those servers immediately, but rather consults the list in the event of a failover, using it to decide who to connect to next.

The entire process is essentially the art of using a statically maintained list to bootstrap getting the more valuable dynamically maintained list.

### Server Configuration

In the server this list can be specified via the conf/multipoint.properties file like so:

```
server      = org.apache.openejb.server.discovery.MultipointDiscoveryAgent
bind        = 127.0.0.1
port        = 4212
disabled    = false
initialServers = 192.168.1.20:4212, 192.168.1.30:4212, 192.168.1.40:4212
```

The above configuration shows the server has an port 4212 open for connections by other servers for multipoint communication. The initialServers list should be a comma separated list of other similar servers on the network. Only one of the servers listed is required to be running when this server starts up — it is not required to list all servers in the network.

### Client Configuration

Configuration in the client is similar, but note that EJB clients do not participate directly in

multipoint communication and do not connect to the multipoint port. The server list is simply a list of the regular ejbd:// urls that a client normally uses to connect to a server.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.RemoteInitialContextFactory");
p.put(Context.PROVIDER_URL,
"failover:ejbd://192.168.1.20:4201,ejbd://192.168.1.30:4201");
InitialContext remoteContext = new InitialContext(p);
```

Failover can work entirely driven by the server, the client does not need to be configured to participate. A client can connect as usual to the server.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.openejb.client.RemoteInitialContextFactory");
p.put(Context.PROVIDER_URL, "ejbd://192.168.1.20:4201");
InitialContext remoteContext = new InitialContext(p);
```

If the server at 192.168.1.20:4201 supports failover, so will the client.

In this scenario the list of servers used for failover is supplied entirely by the server at 192.168.1.20:4201. The server could have aquired the list via multicast or multipoint (or both), but this detail is not visible to the client.

### Considerations

#### Network size

The general disadvantage of this topology is the number of connections required. The number of connections for the network of servers is equal to (n * n - n) / 2, where n is the number of servers. For example, with 5 servers you need 10 connections, with 10 servers you need 45 connections, and with 50 servers you need 1225 connections. This is of course the number of connections across the entire network, each individual server only needs n - 1 connections.

The handling of these sockets is all asynchronous Java NIO code which allows the server to handle many connections (all of them) with one thread. From a pure threading perspective, the option is extremely efficient with just one thread to listen and broadcast to many peers.

#### Double connect

It is possible in this process that two servers learn of each other at the same time and each attempts to connect to the other simultaneously, resulting in two connections between the same two servers. When this happens both servers will detect the extra connection and one of the connections will be dropped and one will be kept. In practice this race condition rarely happens and can be avoided almost entirely by fanning out server startup by as little as 100 milliseconds.

**Recommandation**

As mentioned the initialServers is only used for bootstrapping the multipoint network. Once running, all servers will dynamically establish direct connections with each other and there is no single point of failure.

However to ensure that the bootstrapping process can occur successfully, the initialServers property of the conf/multipoint.properties file must be set carefully and with a specific server start order in mind. Each server consults its initialServers list exactly once in the bootstrapping phase at startup, after that time connections are made dynamically.

This means that at least one of the servers listed in initialServers must already be running when the server starts or the server might never become introduced and connected to all the other servers in the network.