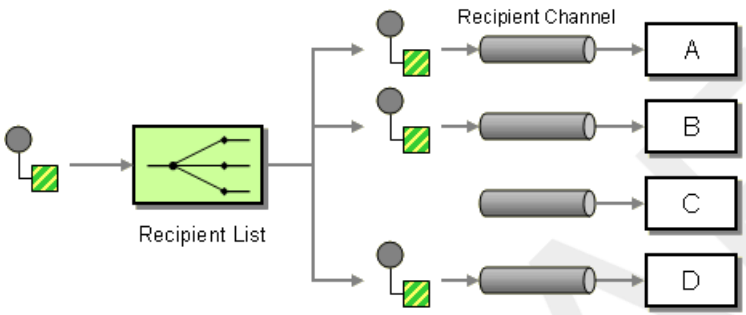
Multicast

Overview

The *multicast* pattern is a variation of the recipient list with fixed destinations pattern, which is compatible with the *InOut* message exchange pattern (in contrast to recipient list, which is only compatible with the *InOnly* exchange pattern).

Figure 22. Multicast Pattern



Multicast with a custom aggregation strategy

Whereas the multicast processor receives multiple *Out* messages in response to the original request (one from each of the recipients), the original caller is only expecting to receive a *single* reply. There is thus an inherent mismatch on the reply leg of the message exchange. In order to overcome this mismatch, you must provide a custom *aggregation strategy* to the multicast processor. The aggregation strategy class is responsible for aggregating all of the *Out* messages into a single reply message.

Consider the example of an electronic auction service, where a seller offers an item for sale to a list of buyers. The buyers each put in a bid for the item and the seller automatically selects the bid with the highest price. You can implement the logic for distributing an offer to a fixed list of buyers using the `multicast()` DSL command, as follows:

```
from("cxf:bean:offer").multicast(new HighestBidAggregation
Strategy()).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buy
er3");
```

Where the seller is represented by the endpoint, `cxf:bean:offer`, and the buyers are represented by the endpoints, `cxf:bean:Buyer1`, `cxf:bean:Buyer2`, `cxf:bean:Buyer3`. In order to consolidate the bids received from the various buyers, the multicast processor uses the aggregation strategy, `HighestBidAggregationStrategy`. You can implement the `HighestBidAggregationStrategy` in Java, as follows:

```
// Java
import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.apache.camel.Exchange;

public class HighestBidAggregationStrategy implements AggregationStrategy {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        float oldBid = oldExchange.getOut().getHeader("Bid", Float.class);
        float newBid = newExchange.getOut().getHeader("Bid", Float.class);
        return (newBid > oldBid) ? newExchange : oldExchange;

    }
}
```

Where it is assumed here that the buyers insert the bid price into a header named, `Bid`. For more details about custom aggregation strategies, see Aggregator on page 61.

Parallel processing

By default, the multicast processor invokes each of the recipient endpoints one after the other (in the order listed in the `to()` command). In some cases, this might give rise to unacceptably long latency. To avoid such long latency times, you have the option of enabling parallel processing in the multicast processor by passing the value `true` as the second argument. For example, to enable parallel processing in the electronic auction example, you could define the route as follows:

```
from("cxf:bean:offer").multicast(new HighestBidAggregationStrategy(), true).
    to("cxf:bean:Buyer1", "cxf:bean:Buyer2", "cxf:bean:Buyer3");
```

Where the multicast processor now invokes each of the buyer endpoints in a separate thread.