Resource-Oriented Concurrent Processing

Jeremy Deane

A fundamental shift in the IT industry has occurred; Amdahl's law has usurped Moore's law. Because the number of transistors on a chip is not doubling every 24 months, developers can no longer "scale up" simply by adding faster processors. Rather, developers must now approach the development of applications with concurrency in mind. And that is because the chip industry is now doubling the number of cores every 24 months. The primary challenge to developers is breaking a problem down so that it can be handled concurrently. The other challenge is to implement a concurrent solution on the Java Platform.

A typical batch order process illustrates the problem. A web service is created to process an XML file, Listing DEA-1, containing a batch of orders. The XML is converted into a domain model object (DMO) and then the DMO is passed to a service responsible for processing the orders within the batch.

```
<orders batchId='A1234' date='07252010' xmlns="cogito.online.domain">
    <order id='X1121' customer='Copperfield' item='Dice' amount='2'/>
    <order id='X1122' customer='Copperfield' item='Top Hat' amount='1'/>
```

Listing DEA-1

Listing DEA-2 will process all the orders but within the same thread. This single threaded approach precludes the order process from "Scaling Up" by adding more cores.

```
Public class BatchServicesImpl implements BatchServices {
  public void synchronousProcessing(List<Order> orders) throws Exception {
    for (Order order : orders) {
          OrderUtility.process(order);
  }
```

Listing DEA-2

Fortunately, the Java Concurrent Package provides a mechanism for processing the orders in parallel. In Listing DEA-3, a Thread Pool is created which acts as a throttle preventing an out of memory exception. Second, each order is processed in its own thread allocated from the pool. This approach is similar to the *Fire and Forget* message exchange pattern.

```
Public class BatchServicesImpl implements BatchServices {
  private Executor pool = new ThreadPoolExecutor(10, 50, Long.MAX_VALUE,
                          TimeUnit.SECONDS,
```

```
                        new LinkedBlockingQueue<Runnable>(2000),
                        new ThreadPoolExecutor.DiscardOldestPolicy());

  public void traditionalProcessing(List<Order> orders) throws Exception {
    for (Order order : orders) {
      Object[] constructorArguments = {order};
            OrderProcess orderProcess = (OrderProcess) DomainModelFactory.
              getFactory().getBean("orderProcess", constructorArguments);

          pool.execute(new Thread( orderProcess ));
  }
```

Listing DEA-3

The Scala Actors Package provides another mechanism for processing the orders in parallel. Scala is a hybrid functional/object language that integrates with Java and runs in the Java Virtual Machine (JVM). Scala Actors provide event based processing of asynchronous messages. Thus, in Listing DEA-4, each order (message) is passed to an Actor for processing (event).

```
Public class BatchServicesImpl implements BatchServices {
  public void functionalProcessing(List<Order> orders) throws Exception {
    FunctionalManager functionalManager =
      (FunctionalManager)DomainModelFactory.
      getFactory().getBean("functionalManager");
    functionalManager.start();
    for (Order order : orders) {
      functionalManager.$bang(order);
    }

  class FunctionalManager(jPrices:java.util.Map[String, String],
       jDiscounts:java.util.Map[String, String]) extends Actor with LogHelper {
       def act() {
         loop {
           react {
            case order: Order => processOrder(order)
           }
         }
       }

       def processOrder (order:Order):Unit = {
         //get the price and calculate sub-total
```

```
        val price = prices.getOrElse(order.getItem(), "0").toDouble;
        val subTotal = order.getAmount() * price

        //get the discount and apply
       val discount = discounts.getOrElse(order.getItem(), "0").toDouble;
       if (discount > 0) {
         val percentage = 1 - discount
         logOrderSummary(order, price, discount, subTotal, subTotal*percentage)
       } else {
        logOrderSummary(order, price, discount, subTotal, subTotal)
       }
     }
```

Listing DEA-4

Both the Java Concurrent and the Scala Actors solutions concurrently execute subtasks in parallel allowing order processing to "Scale Up" as more cores are added. However, the Java solution requires a developer to deal with complexity of multithreaded execution while the Scala solution requires a developer to learn about functional programming. Fortunately for the developer, there is another approach, resource-oriented concurrent processing.

Resource Oriented Architecture (ROA) extends the REST architectural style but provides a deeper, more extensible and transport independent foundation. While RESTful web services require the use of HTTP, resource-oriented services support additional transports such as JMS or FTP.

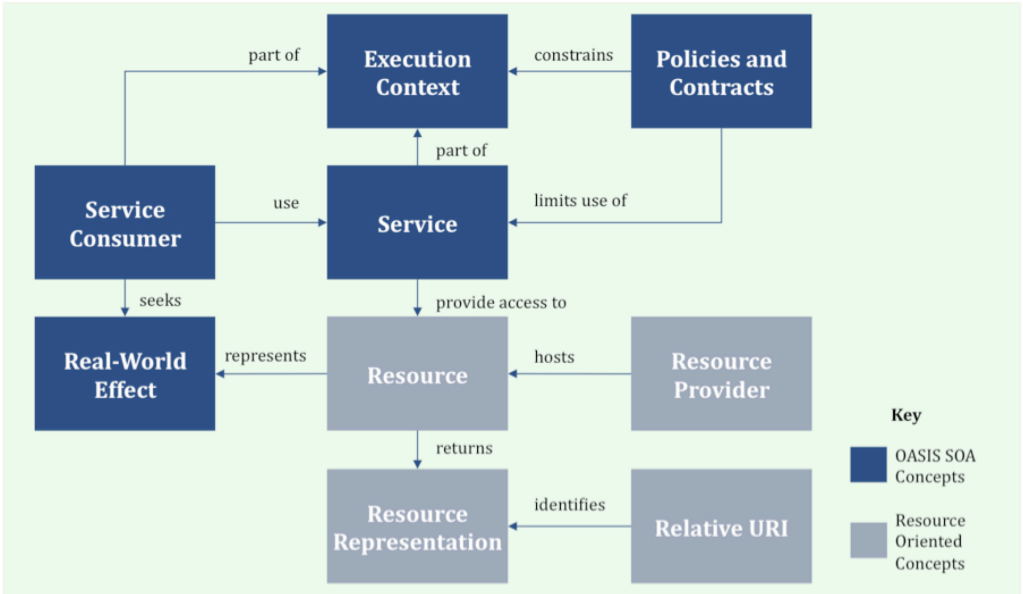

Figure DEA-1 – ROA Conceptual Model

NetKernel is a Resource Oriented Computing (ROC) platform that runs on the JVM. The core of resource-oriented computing is the separation of logical requests for information (*resources*) from the physical mechanism (*code*), which delivers the requests. A resource is identified by a Universal Resource Identifier (URI) and is managed within a logical address space. A REST-based micro-kernel handles all requests for resources, resolving the URI to the address space and returning a representation of that resource.

The resource-oriented order processing solution defines an `Orders` resource that creates N `Order` resources. A client submits an XML representation of the `Orders` resource to *http://localhost:8080/ cogito/online/batch/12345* and the micro-kernel resolves that logical request to physical code. `OrdersAccessor`, in Listing DEA-5, is a Java class implementing the NetKernel API, but could have been implemented in any language (e.g. Groovy or Scala) that runs on the JVM. The `OrdersAccessor` then submits a sub-request for each order.

```
public class OrdersAccessor extends StandardAccessorImpl {
  private void handleBatch(INKFRequestContext context) throws Exception {
    //for each order, issue an async request
    for (int i=0; i < ordersNodes.getLength(); i++) {
      DOMXDA orderDOM = new DOMXDA(ordersDOMXDA.getFragment(
                                   ordersNodes.item(i)),false);
      String orderID = orderDOM.getText("/order/@id", true);

      //issue sub-request - Fire and Forget
      INKFRequest subRequest = context.createRequest
                              ("res:/cogito/online/order/" + orderID);
      subRequest.setVerb(context.getThisRequest().getVerb());
      subRequest.addPrimaryArgument(orderDOM);
      context.issueAsyncRequest(subRequest);
    }
  }
}
```

Listing DEA-5

The `Order` representation submitted internally to *res:/cogito/online/order/X1121* resolves to the same Java class, `OrdersAccessor`, but is processed by a different method, `handleOrder (Listing DEA-6)`. The `handleOrder` method makes sub-requests for pricing and discount information. The representations returned from those requests are cached by the micro-kernel. Thus, the internal request *res:/cogito/online/price/Dice* only resolves to physical code once.

```
public class OrdersAccessor extends StandardAccessorImpl {
  private void handleOrder(INKFRequestContext context) throws Exception {
```

```
        //1. lookup the price
        String item = orderDOMXDA.getText("/order/@item", true);

        Double price = retrievePricingInformation(context,
             "res:/cogito/online/price/" + item.replaceAll("\\W",""));

        //2. calculate the base charge
        Double subTotal = price * new Double(orderDOMXDA.getText
                                          ("/order/@amount", true)) ;
        //3. lookup discount and apply
        Double discount = retrievePricingInformation(context,
             "res:/cogito/online/discount/" +item.replaceAll("\\W",""));

        Double charged=0.00;

        if (discount > 0) {
             charged = subTotal * (1-discount);
        } else {
             charged = subTotal;
        }
  private Double retrievePricingInformation (INKFRequestContext context,
                                             String uri) throws Exception
  {
       INKFRequest request = context.createRequest(uri);

       /*
        * issue sync request; under the covers this is an async
        * request-response
        * via the kernel.
        */
       return (Double) context.issueRequest(request);
  }
}
```

Listing DEA-6

Unfortunately not every process can be executed concurrently using *Fire and Forget*. For instance, the process may need to *fork* a long running subtask, continue processing the current task, and then *join* back at a later point to get the result from the forked subtask. For this more complicated case both Java and Scala provide constructs called *Futures*. NetKernel provides this capability via its API.

In Listing DEA-7, the taxed order process forks and joins on several subtasks. In addition, the parent process returns a total amount charged when processing of all the orders is complete. Within the taxed order process, *asynchronous sub-requests* are issued to the micro-kernel for price, discount and tax values. *Handles* to the future values are returned by the micro-kernel. Processing continues and, as needed, the main process joins to the future values provided by the handles.

```
public class TaxedOrdersAccessor extends StandardAccessorImpl {
  private void handleBatch(INKFRequestContext context) throws Exception {

  /*
   * issue sync request; under the covers this is an async
   * request-response via the kernel.
   */
    Double orderTotal = (Double)context.issueRequest(subRequest);
    ordersTotal = ordersTotal + orderTotal;

    AccessorUtility.returnMessage(context, "Total Batch Charges: $"
                                          + ordersTotal.intValue());
  private void handleOrder(INKFRequestContext context) throws Exception {
    //1. lookup the price, tax and discount in parallel
    String item = orderDOMXDA.getText("/order/@item", true);
    String customer = orderDOMXDA.getText("/order/@customer", true);

    INKFAsyncRequestHandle priceHandle = requestPrice(context, item);
    INKFAsyncRequestHandle discountHandle = requestDiscount(context, item);
    INKFAsyncRequestHandle taxHandle = requestTax(context, customer);

    //2. Join back on the price request (within 200ms)
    //   and calculate base charge
    Object price = priceHandle.join(200);

    //price request took too long
    if (price == null) {
      throw new Exception ("Unable to retrieve price for " + item);
    }

    Double subTotal = (Double)price * new Double(orderDOMXDA.getText
                                                ("/order/@amount", true));

    //3. Join back on the discount request within 200ms and apply discount
```

```
    Object discount = discountHandle.join(200);

    //discount request took too long
    if (discount == null) {
      throw new Exception ("Unable to retrieve discount for " + item);
    }

    Double discountedTotal = 0.00;

    //discount gets applied to subTotal
    if ((discount != null) && ((Double)discount > 0)) {
      discountedTotal = subTotal * (1-(Double)discount);
    } else {
      discountedTotal = subTotal;
    }

    //4. Join back on the tax request within 200ms and calculate the tax
    Object tax = taxHandle.join(200);

    //tax request took too long
    if (tax == null) {
      throw new Exception ("Unable to retrieve tax for " + customer);
    }

    Double taxedTotal = discountedTotal * (1 + (Double)tax);
    context.createResponseFrom(taxedTotal);

  private INKFAsyncRequestHandle requestPrice (INKFRequestContext context,
                                               String item) throws Exception
  {
    String priceURI = "res:/cogito/online/price/" + item.replaceAll("\\W","");
    return context.issueAsyncRequest (context.createRequest(priceURI));
  }
}
```

Listing DEA-7

The benefits of implementing concurrent processes on a resource oriented computing platform include multi-core scaling, caching and simplicity. In fact, services built using ROC have proven to be small, simple, flexible and require less code compared to other technology platforms. Furthermore, the platform's advanced caching reduces the amount

of computation required to process duplicate requests. Finally, because all internal requests to the microkernel are asynchronous, processing scales linearly as more *cores* are added.

The demos referenced in this article can be downloaded from *http://files.me.com/jtdeane/j6q6xy*. The enclosed instructions describe how to deploy the applications, run the JMeter tests, and compile the source code using Maven.

References

Java Concurrency in Practice by Brian Goetz

Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine by Venkat Subramaniam

Introduction to Resource Oriented Architecture White Paper by 1060 Research

About the Author

Jeremy Deane
Director of Research & Architecture
Plymouth Rock Assurance

Jeremy Deane has over 15 years of software engineering experience in leadership positions. His expertise includes Enterprise Integration Architecture, Web Application Architecture, and Software Process Improvement. In addition, he is an accomplished speaker and technical author.

jeremy.deane@gmail.com