Lightweight grids with Terracotta

Cesario Ramos & Peter Veentjer
Xebia

Introduction

Grid architectures are the latest evolution of the distributed computing paradigm. During the 90-ties the grid approach of computation became widely accepted as a new approach to distributed computing. This approach focuses on ways for large-scale resource sharing and how to cope with distributed computational challenges. Within the Java community a number of products have emerged for building your own grid based architectures. These products tend to focus more on providing availability and scalability solutions for enterprise systems. One of them is the open source product Open Terracotta. Open Terracotta provides a unique way for developing distributed Java applications.

In this article we will discuss how Terracotta can help in developing grid based applications. We will start with a short overview on grids and sketch the problem domain where Open Terracotta could be used. After that we introduce Open Terracotta and discuss an example of a simple pipeline system build with Open Terracotta.

What is a Grid?

Literature provides many different definitions for grids. The various definitions all have in common that grids are about the distribution of data, logic and the sharing of resources.
The term grid is mostly known in the context of a power grid. In a power grid a number of power stations and substations are interconnected by a transmission circuit. All power stations together are able to handle the required power consumption. Also the grid is able to cope with failure of individual power stations by having redundant paths between power stations and substations. To the individual consumer the power grid appears as one big power resource.
Within the context of computation a grid is a number of nodes connected by a network that appears to consumers (users or applications) as one big computing resource. Issues such as availability, scalability and performance are addressed in a grid.

An application executing on a grid can have its processing requirements distributed over various nodes. The grid takes care of **dynamic allocation of resources** and the **distribution of its processing requirements**. The dynamic allocation makes it easy to scale out by just adding more machines and letting the grid allocate and manage them. By distributing its processing logic the application can benefit from parallelization. This type of grid is known as a **computational grid** where machines provide processing power to cope with high workloads
Although a computational grid works fine for computation intensive problems for more day to day applications this increase in computing power is bounded by the application's data access speed. What you could do is letting the processing logic execute on a node that has the data locally available. In addition to fast data access such an approach also

opens up possibilities for reducing load on storage devices i.e. databases and eliminating them as a Single Point Of Bottleneck (SPOB). A grid where distributed logic is co-located with the data it needs using distributed caching techniques is known as a **data grid**.

What problems do grids solve?

In everyday application development a grid based approach is usable in situations where you need high availability, scalability and performance, but the client/server architecture and distributed programming model gets in your way.
The client server architecture as depicted in the figure below has as major disadvantage that as the number of clients increase so will the load on the server as well as the bandwidth needed to serve all client requests. The scaling of clients is therefore limited as the server acts as a SPOB. What can you do to make your system scalable? One obvious thing you can do is to **avoid a SPOB** and spread load across multiple resources.

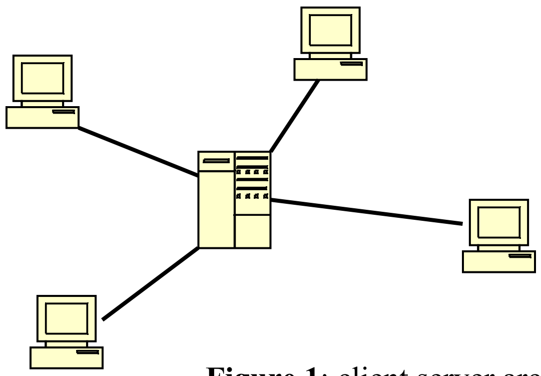

Figure 1: client server architecture

The central server also imposes an availability risk because it is a SPOF. How can you make your system be down for the least possible time and have your system up and running correctly for the longest possible time? One obvious thing you can do is to **avoid a SPOF** and introduce redundancy in machinery and replication of components.
In a grid the possibility to spread load across multiple nodes and replicate components is often offered as a first class service. The grid middleware takes care of distributing your application logic and/or data according to your specific configuration enabling failover to other nodes in case of failure. The grid middleware can also adopt new nodes into the grid dynamically using the newly available resources in future processing.
Using the grid middleware also simplifies your distributed programming model. Using a RPC style for your system leaves you with lots of 'objects' and complex communication paths. Somehow these objects must know about each other and synchronize their activities. By using a grid approach the communication complexity can be dramatically reduced. There is a common shared memory where objects can write and read from. In grid architecture the middleware itself takes care of synchronization and distribution of data and events. The programming model is therefore much simpler from a developer's perspective.

Within the context of enterprise systems when we cluster our machines we usually setup a load balancer in front of clustered web and app servers and have them access the database as shown in the figure below.

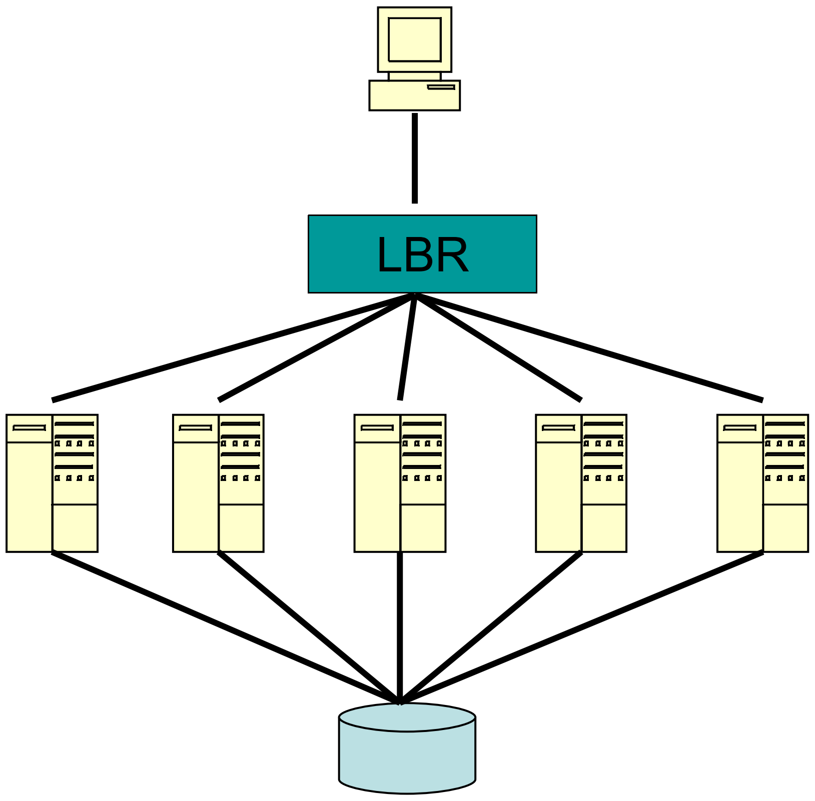

Figure 2: clustered architecture

When we do this we end up with lots of things to worry about. Because we are now running a clustered application we have to worry about distributing state across nodes. In case of a web application we have to take into account the size of our HttpSession and the impact on network and CPU utilization for serializing/deserializing it. What you often see is that the database is used to store this conversational data. Then when a node fails and a request is redirected to another node it can just read the conversational data from the database and proceed. In doing this, we unnecessary push work and responsibility to the database. Now when we start to scale our systems and the load increases the demands on the database increases with it. Eventually the database becomes your SPOB and SPOF and then you have the challenge of clustering your database. This is usually not only more complicated but also more expensive than clustering at the application server level.

What is Open Terracotta?

Open Terracotta see http://www.terracotta.org/ provides a platform for realizing scalability and availability by providing clustering at the JVM level. It reduces the complexity of distributed computing and provides a basis for realizing grid solutions. It provides a distributed shared memory that can be accessed as if it was local memory by multiple JVMs. This approach allows you to write Java applications for a single JVM and have it distributed over various JVMs in a transparent way.

Terracotta extends the local heap space of your JVM by keeping a persistent virtual heap on an external machine. The external heap is realized by a Terracotta server and its size is only bounded by the disk size of the server. Using the external heap is completely transparent; you do not have to do anything special. It works much like paging in your OS when faulting and flushing memory pages from disk to main memory and vice versa. Objects that can reside in this external heap are called Distributed Shared Objects (DSO). A DSO is an object that has been instrumented by Terracotta so that modification of its fields and execution of synchronization operations on it i.e. wait/notify are synchronized across the cluster. It has a unique cluster wide id just like you'd expect for a single JVM. Terracotta needs a starting point to keep track of the DSOs. For this they introduced the concept of a shared root that acts as the starting point of a distributed object graph. All objects that are part of this graph become a DSO. A shared root is persistent and will never be removed from the Terracotta server. (If you choose to run the Terracotta server in persistent mode it will also survive server failures.) Any DSO that is part of the graph is also persisted until explicitly removed from the external heap by the application logic. Shared roots are only instantiated once and then moved to the Terracotta server. When a root is created again its data is faulted in from the Terracotta server.

For Terracotta to be able to synchronize any changes you make to a DSO, changes have to be made within the context of a Terracotta transaction. A Terracotta transaction is started when a lock is acquired and ended when the lock is released. The changes are flushed to the server and then faulted in at the JVMs that need it and when they need it. A JVM needs the change when it has objects instantiated on its heap that are modified in the transaction. The transmission of updated data is done using fine grained replications. What this means is that only the actual changes of an object are moved around and not the entire object or object graph as would be the case when using Java serialization.

A simple producer consumer example

Let's look at a simple producer consumer example and how to realize it with Terracotta. Below we see an excerpt from a producer and a consumer Java class. You can see that they both have a queue that they use. The producer puts its produced items in this queue and blocks when the queue is full. The consumer takes the items form the queue and blocks when the queue is empty.

```
public class Consumer {
      private LinkedBlockingQueue<Something> queue;
      public void consume() {
            Something msg = null;
            while(keepConsuming) {
            msg = queue.take()
      …
```

```
public class Producer {
      private LinkedBlockingQueue<Something> queue;
      public void produce() {
            Something msg = …;
            while(keepProducing) {
            msg = queue.put(msg);
      …
```

Figure 3: producer, consumer example code

For this to work the producer and consumer must use the same instance of the queue. This is where Terracotta comes in. With Terracotta we can declare the queue as a shared root and have it distributed across the cluster. We can tell Terracotta to make the queue a shared root by specifying it in a configuration file as you can see below.

```
...
<instrumented-classes>
 <include>
  <class-expression>
    example.Something
  </class-expression>
...
<root><field-name>
      example.Consumer.queue
 </field-name>
 <root-name>rootQueue</root-name>
</root>
<root>
 <field-name>example.Producer.queue</field-name>
 <root-name>rootQueue</root-name>
</root>
...
```

Figure 4: example tc-config for producer consumer.

The above configuration tells Terracotta that the queue data member of the Producer and Consumer class are shared roots. It also defines them to be the same shared root by declaring them to have the same root name.
As a result when a producer is started it produced items are placed into the queue that is accessible from a consumer. You can start as many producer and consumers as you want in various JVMs and they will all communicate and synchronize using the shared root.

Availability and scalability using Terracotta

Availability is offered by ensuring that distributed state is persistent. State that is written to the Terracotta server is persistent until explicitly removed. Furthermore the terracotta server itself can be clustered and have its state persisted to disk. There is no SPOF. In case of client failure and a request is rerouted, the client processing the request gets the required data from the Terracotta server or from main memory depending on the situation. This idea is the same as storing data in a database or disk for achieving availability. The difference is that storing it on the Terracotta server is much faster. You do not need any mapping between data i.e. ORM, row mapping or even serialization.

Scalability can be realized by partitioning your data. You can implement your own partitioning logic that routes work to the data it needs. But Terracotta also does some automatic partitioning of your data. The Terracotta server will only push data to a client when the client needs it. This means that if clients work on a specific set of data automatic partitioning occurs and scalability is not bounded by the clients' heap space. When a client runs low on heap space a least recently used eviction policy is used to free client heap memory. When you add a client the total client heap size increases. You can also handle more requests as the new client works on its own data set. As mentioned above Terracotta uses fine grained replication to keep the network load at a minimum.

Example Pipeline application

Let’s assume that we need to export private information to third parties for them to be able to perform various calculations. Because you may not provide this private information directly to third parties the information needs to be transformed in such a way that no information needed for the calculations is lost and that there is no way of tracing information back to its origin. In order to do this we need to transform these very large files from one representation to another within a certain time limit.
We are going to realize this system using Terracotta and are also going to use the Prometheus framework see http://prometheus.codehaus.org/. Prometheus is an open source concurrency library that extends java.util.concurrent. It helps you to separate your business logic from your concurrency logic so that you can focus solely on your problem and not worry about basic concurrency issues.
Prometheus provides you coarse grained building blocks for developing concurrent applications. For this example we only use a limited set of the available components. We will use a Processor, Process, Channel and a Repeater. These components will be discussed throughout the text.

The first thing we need to do in order to develop this application is to parse the content of the file, then we need to put the individual records through a number of CPU intensive transformations and ultimately the transformed records need to be written to disk in the their original sequence. For the example we assume that we need four processes. A parse process for reading the files and a write process for writing the outcome back to storage. Instead of the CPU intensive transformation processes lets say for simplicity that we have a Fib process for calculating Fibonacci numbers and a Pi process for calculating pi at various precisions.
A very naïve approach to solving this problem would be to use sequential processing as depicted in the figure below.

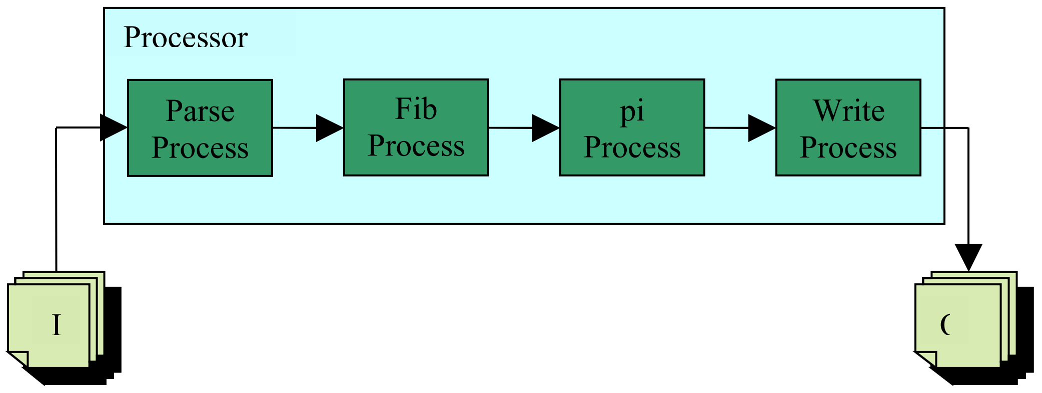

Figure 5: Sequential processing
→ Dataflow

What we see here is a processor that contains the four processes. A processor is an abstraction defined within Prometheus. The processor provides an execution environment for a process. It takes care of handling incoming and outgoing messages and provides queuing, exception handling and event dispatching services. A process is used to define your application logic. Here you handle your messages and focus on producing, transforming and consuming of messages etc, whatever is needed for your specific application. For a process a POJO could be created within Spring for example. For use within Prometheus you only have to respect the Prometheus contract being that you have to implement a receive method.

For e.g. the Fibonacci (Fib) process we have the following code:

```
public class FibonacciProcess {

      …

    public void receive(Task task) {
        task.setOutFibonacci(fibonacci(task.getInFibonacci()));
        log.info(task);
    }

    public static long fibonacci(long n) {
        if (n <= 1)
            return n;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Figure 6: FibonacciProcess source code.

As you can see the FibonacciProcess is completely separated from any concurrency and environmental logic, it only contains core application logic. This approach makes it possible to reuse the process in various distributed configurations as you'll see later.
The receive method of the FibonaciProcess is called by the processor with a Task as an argument. From the task it gets the Fibonacci number to calculate and then stores the result in the task again. After that the modified Task object is returned to the Processor which takes care of further routing it to the pi process.

In the sequential processing solution depicted above, the records are processed one at a time. The parse process reads a record then the processor routes the record to the fib process for the first transformation. After the fib process finishes the processor routes the transformed record the pi process for the second transformation etc.
In this solution you cannot take advantage of multiple machines or even one multi core system. You also do not exploit idle CPU cycles while the write or parse process are waiting on I/O. So even on a single core system you could be wasting a lot of system resources. You can also not scale horizontally, because you can't distribute your processing, except at the highest level where you could duplicate the entire system.

In order to improve the performance you can separate the processors and processes to be able to execute them independently. This gives you a setup as depicted in the figure below. Here you see the each process executes within the context of a single processor. We see that the Fib process has N and the Pi process M worker threads. All these threads consume records from the channels in parallel for processing. The processors are linked by channels (implemented by util.concurrent.LinkedBlockingQueue) where each process reads from and writes to.
The use of multiple threads causes that the order of processing is no longer guaranteed so we need to add a resequencer, see Pattern from Enterprise Integration Patterns, to restore the ordering before writing to file by the write process. Reordering of messages in the case of this application takes place based on the creation time of the message being passed.

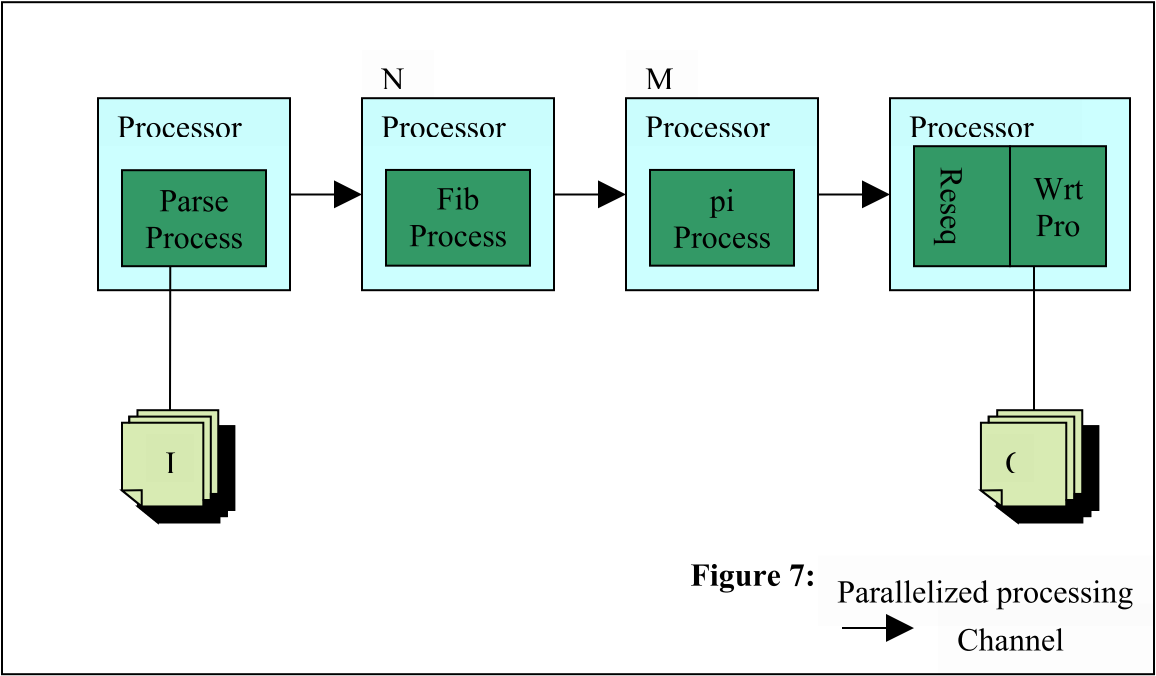

Figure 7: Parallelized processing

To easily configure the various processors, processes and channels we use Spring. Below you see the Spring configuration file for the FibonacciProcess. You see that we configure a StandardProcessor with the fibonacciProcess as well as the input (fibonacci-Processor-input) and output (fibonacciProcessor-output) channels. The input channel connects the ParseProcess with the FibProcess and the ouput channels connect the FibProcess with the PiProcess.

```
    <bean id="fibonacciProcess"
          class="example.processes.FibonacciProcess"/>

    <bean id="fibonacciProcessor"
          class="org.codehaus.prometheus.processors.standardprocessor.S
tandardProcessor"
```

```
          parent="abstractProcessor">
        <constructor-arg index="0"
                         ref="fibonacciProcessor-input"/>
        <constructor-arg index="1"
                         ref="fibonacciProcess"/>
        <constructor-arg index="2"
                         ref="fibonacciProcessor-output"/>
    </bean>
```

Figure 8: Spring configuration of FibonacciProcess.

Now our system is ready to be tuned for optimal performance on a single machine. The next thing we need to do is distribute it over various machines so we can increase performance but also availability and reliability. As mentioned above, one way of ensuring reliability and availability is to introduce redundancy in machinery and replication of processing components.
Terracotta can help us to realize this in a relatively easy way. As we already have a correct concurrent program, we can use Terracotta to distribute the Processors and their Processes across various machines. The channels are the obvious candidates for putting under Terracotta management. So we define them as shared roots.
All records pass thru the channels. So the threads running on a specific JVM can consume as many records as that particular JVM can handle.
In order to put the channels into Terracotta space we can put all the channels into a Spring context and have Terracotta distribute that. Below we can see the Spring channels configuration. What we see is that pipe2 is named ‘piProcessor-output’ and ‘fibonacciProcessor-input’ and therefore acts as the pipe between the FibonacciProcess and the PiProcess.

```
    <bean id="pipe1"
          name="piProcessor-input,parserProcessor-output"
          class="org.codehaus.prometheus.channels.BufferedChannel">
        <constructor-arg index="0" value="100"/>
    </bean>
    <bean id="pipe2"
          name="piProcessor-output,fibonacciProcessor-input"
          class="org.codehaus.prometheus.channels.BufferedChannel">
        <constructor-arg index="0" value="100"/>
    </bean>
    <bean id="pipe3"
          name="fibonacciProcessor-output,fileWritingProcessor-input"
          class="org.codehaus.prometheus.channels.BufferedChannel">
        <constructor-arg index="0" value="100"/>
    </bean>
```

Figure 9: Spring configuration of pipeline.

In order to have Terracotta distribute these pipes we have to define a terracotta configuration file. We have to declare which Spring beans to cluster, declare the Task object as a DSO and define locking semantics see http://www.terracotta.org/ for more info on that. Below an excerpt of the Terracotta configuration file.

```
…
      <spring>
            <jee-application name="*">
                <instrumented-classes>
                    <include>
                        <class-expression>example..*</class-expression>
                    </include>
                </instrumented-classes>
                <application-contexts>
                    <application-context>
                        <paths>
                           <path>applicationcontext-channels.xml</path>
                        </paths>
                        <beans>
                            <bean name="pipe1"/>
                            <bean name="pipe2"/>
                            <bean name="pipe3"/>
                        </beans>
                    </application-context>
                </application-contexts>
                  …
            </jee-application>
        </spring>
            …
```

Figure 10: tc-config pipeline

With this in place we can now startup multiple Fibonacci and Pi processes and have easily realized a grid solution. The result is depicted in the figure below.

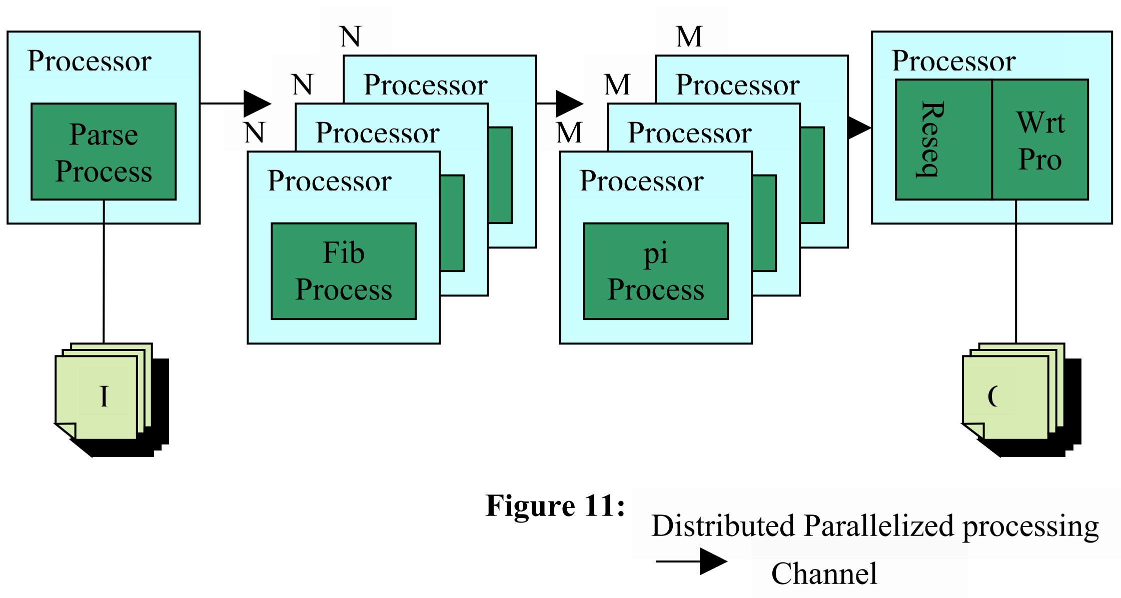

Figure 11:

Conclusion

We just created a distributed multithreaded application with high availability and scalability characteristics without writing a single line of concurrent code. No use of java.util.concurrent, no java.lang.Thread, no nothing of that kind. All of this hard to code issues are abstracted away by using Prometheus and Terracotta. Furthermore we have transparent distribution of processing logic across different nodes, nodes can come and go transparently to the grid and the system keeps on running. What we have shown here is how easy it is with Terracotta and Prometheus to create a lightweight grid.