4
Implementation

The *Data Processing Slice Load Balancing Solution* presented in chapter 3 was implemented as a prototype using the Java programming language [51] version 7 and CoreDX DDS [52] version 3.5.3. The Java language was chosen because Java programs can run on many different computer architectures and operating systems; it is one of the most popular programming languages, well adopted both at academia and industry; and SDDL (section 2.2) is also implemented on this programming language. The reason for choosing CoreDX DDS as the communication layer was because it delivers high-performance communication and low-overhead, and has shown low latency and high throughput message delivery numbers [52]. CoreDX DDS takes account of fundamental design principles aimed to meet the requirements of real-time and near-real time systems, including minimal data copies, compact encoding on the wire, light-weight notification mechanisms, pre-allocation of resources and pre-compilation of type-specific code blocs [21]. It also can run on many computer architectures and operating systems such as x86, x86_64, ARM, i686pc, sun4u architectures, and Windows, Linux, Android and Solaris operational systems, respectively.

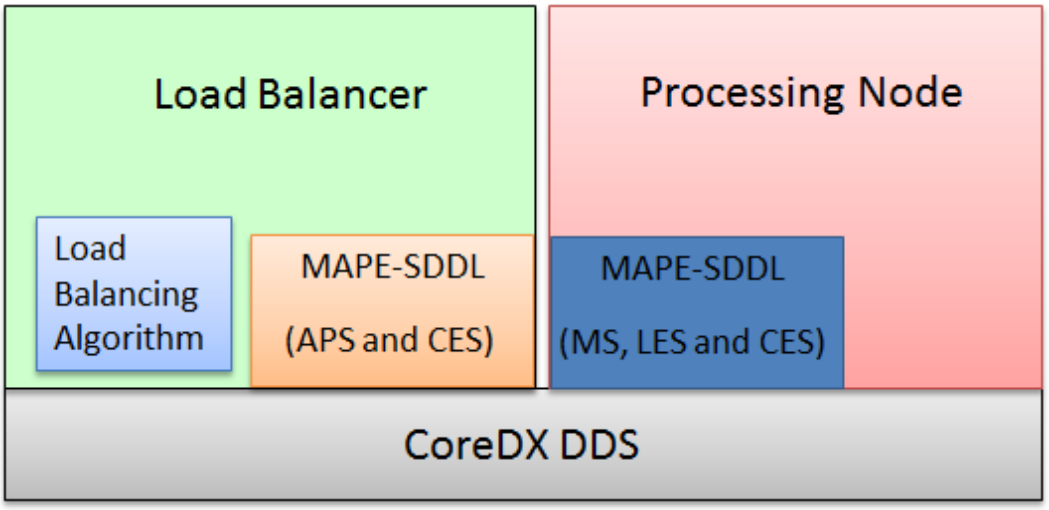

Figure 16 – Implementation architecture

Figure 16 shows the main modules that are used in the PNs and the Load Balancer of the DPSLB mechanism. As mentioned in section 4.1, it is based on the Autonomic Computing Framework MAPE-K adapted to the SDDL middleware (MAPE-SDDL). The services of MAPE-SDDL exchange control and monitoring data (e.g. Action Plan and Monitoring Event Notification). PNs also use CoreDX DDS to receive/send data from/to Client Nodes and for sending their caches to other PNs. The Load Balancer further has a module, called Load Balancing Algorithm, which will be described in section 4.2.

4.1 MAPE-SDDL

MAPE-SDDL was developed cooperatively between LAC at PUC-Rio and the *Laboratório de Sistemas Distribuídos* (LSD) at the Federal University of Maranhão (UFMA). It should be noted that MAPE-SDDL was neither proposed nor developed as part of this thesis, but only used to facilitate the implementation of the DPSLB prototype.

MAPE-SDDL, which is inspired by the MAPE-K reference model, is a autonomic extension layer of the SDDL that incorporates general dynamic adaptive capabilities into SDDL. Its goal is to support resource monitoring, as well as analysis, planning and execution of dynamic reconfigurations on components of the SDDL middleware. It comprises four services: Monitoring Service (MS), Local Event Service (LES), Analysis and Planning Service (APS), and Control and Executing Service (CES).

The MS collects data from any computing resources within SDDL, such as Gateways or PNs. The monitoring is applied to properties (e.g. CPU load, amount of memory available, network bandwidth and latency or number of *Slices* assigned to each PN) of these resources. Each property is associated with a set of operation ranges, which are defined by the MAPE-SDDL user. For example, one could use the following operation ranges for monitoring the CPU usage: [0%, 30%], [30%, 70%] and [70%, 100%]. The MS then notifies the LES (located at the same node of MS) whenever the monitored property switches its operation range, which might indicate a significant change on resource usage.

The LES receives these range change events from the MS and publishes higher level event notifications to subscribed components. Events are occurrences

which indicate that a resource availability condition changed for a specified period of time, i.e. its duration time. Event evaluation is based on regular expressions written by application developers or operators, as part of each event definition. For an event notification to be triggered, the corresponding expression must remain valid during the specified duration time. This avoids generating events for short-lived situations (e.g. a CPU load peak on a PN during few seconds).

The APS analyzes the received event notifications from LES and makes a diagnosis of the problems to be solved (e.g. load unbalance between PNs). After diagnosis, the APS seeks the dynamic reconfiguration actions to resolve the problem, and then builds an appropriate *Action Plan*. The decision-making for building the plan is defined by the user through the use of rules and a rule processing engine (i.e. the Load Balancer in the DPSLB prototype). Each type of reconfiguration *Action* supported by CES receives an ID. This identifier is included in the *Action Plan* in order to allow the CES instances to know what reconfiguration Actions must be performed. Specifically for Load Balancing, the Action Plan is the sequence of *Actions* to be executed by the PNs that have been selected for moving Slices.

Finally, CES is the adaptation engine that applies the corresponding reconfiguration Actions at the resources in response to their changes (e.g. availability or load variation). The capability of moving *Slices* from a PN to another is one example among the types of dynamic reconfiguration actions supported.

The main advantages of using the MAPE-SDDL to implement the DPSLB prototype are a well-defined model (MAPE-K) and framework that easy the monitoring of the PNs, analyzing/planning of the resources and execution of the Actions. One of the most import feature for the implementation of the DPSLB prototype is the MAPE-SDDL´s capability of controlling the Action executions, which facilitates the Load Balancer synchronizes the PNs while they are in Load Balancing Session.

4.2 Load Balancer

Roughly speaking, the Load Balancer receives *Monitoring Event Notifications* through MAPE-SDDL, analyzes them, possibly generates an Action Plan and sends the corresponding commands through MAPE-SDDL. An Action Plan is

generated and sent in response to a detection of unbalance load. The Load Balancer executes the APS and CES services of the MAPE-SDDL, and only the *Control Service* of the CES since it has no local dynamic reconfiguration to perform. The Control Service does not explicitly interact with the Load Balancer since it transparently controls the remote Actions executed at the PNs.

Each *Action* must be executed in the order that it appears in the *Action Plan*, e.g. if an Action Plan consists of the sequence of Actions [*A*, *C*, *B*], then Action *B* will only be executed after *C*, which in turn will wait until *A* is executed. Hence, the successful execution of each Action, on each PN, will confirmed to CES, which will only be able to send the next Action contained in Action Plan after the previous action has been confirmed.

The Load Balancer holds a mapping that contains all PNs and their last five Monitoring Event Notification. In fact, the number of Monitoring Event Notification kept by LB is a parameter which can be adjusted in the LB to configure the size of historical data. When a new PN is detected by the Load Balancer, it notifies the *Load Balancing Algorithm* about the presence of this new PN. It is believed that this behavior can be useful for many load balancing algorithms and can facilitate the development of new algorithms since developers can focus on the development of new load balancing algorithms. After receiving a Monitoring Event Notification, LB calls the Load Balancing Algorithm in order to analyze the system load and decide if any *Slice* should be moved to another PN. With the result of the Load Balancing Algorithm, the Load Balancer generates the Action Plan and sends it through the MAPE-SDDL to the PNs involved in the Load Balancing Process. As explained, LB executes CES of MAPE-SDDL, which controls the execution of the entire Action Plan, so to guarantee that all Actions are correctly executed by all PNs.

```
public interface LoadBalancingAlgorithm {
  public Collection<SliceMovement> analyzeLoad(
    Map<UUID, ProcessingNodeStructure> processingNodeMap);

  public void
    onNewProcessingNode(ProcessingNodeStructure processingNodeStructure);
}
```

Figure 17 – LoadBalancingAlgorithm interface

Concepts such as Action and Action Plan are completely transparent to the Load Balancing Algorithm. Instead, this module only executes the logic for ana-

lyzing the system load, deciding which will be the *Slice-giving* and *Slice-taking* PNs and how many *Slice* should be moved from the first to the latter. In order to enable this separation of concerns, the Load Balancing Algorithm must implement the *LoadBalancingAlgorithm* interface, shown in Figure 17, that consists of two methods: *onNewProcessingNode()* and *analyzeLoad()*. The method *onNewProcessingNode()* is called when the Load Balancer detects that a new PN arrived in the system and *analyzeLoad()* is called every time that a new Monitoring Event Notification is received from a PN. This last method returns a collection of *SliceMovement* objects, which contains the *Slice-giving* and *Slice-taking* PN and how many *Slices* should be moved to the *Slice-taker*.

The *LoadBalancingAlgorithm* interface allows the Load Balancer to have its Load Balancing Algorithm changed dynamically. Moreover, the Load Balancing Algorithm is a black box for the LB, which is concerned only about managing the PNs, generating and controlling the execution of the Actions in the Action Plan. Hence, through a clear separation of concerns, this interface facilitates the development and integration of new load balancing algorithms into the Load Balancer. It is worth to recall that the Load Balancing Algorithm does not need to manage the PNs and their monitoring resources (e.g. CPU and memory utilization) since this work is done by the Load Balancer. To do so, the Load Balancer provides its mapping of PNs every time the Load Balancer call the *onNewProcessingNode()* method. The *Load Balancing Algorithm* module is set as a parameter of the Load Balancer´s constructor, thus facilitating the deployment of other Load Balancer instances that may use different *Load Balancing Algorithms*.

After receiving the collection of *Slices* that should be moved as result of calling the Load Balancing Algorithm, the Load Balancer generates the Actions that will be executed by the PNs involved in the Load Balancing Process. Each single *SliceMovement* object, unfolds to five Actions, which are lower level abstractions on DPSLB, and are executed by the *Slice-taking* and *Slice-giving* PNs as described in section 3.4. These Actions. for all the necessary Slice Moves, constitute the Action Plan.

4.3 Processing Node

The PN (Processing Node) is the managed resource from the perspective of MAPE-K model used in the DPSLB prototype developed by this thesis. Each PN executes the MS, LES and CES services of MAPE-SDDL, but concerning the latter, only its Executing Service since the Control Service is executed by the Load Balancer. In addition to the data processing, which is intrinsically determined by the application build upon DPSLB, each PN periodically verifies its monitored properties and, depending of their operation ranges, notifies the LES that evaluates these values against the specified expression. Hence, LES eventually sends a Monitoring Event Notification, which holds all monitored data, to the Load Balancer through APS. The current version of this prototype periodically checks the PN´s monitored properties every two seconds and then sends a Monitoring Event Notification to Load Balancer.

The PN was not implemented using DDS' Content Filtered Topics due to CoreDX DDS´ impossibility of updating the Content Filtered Topic´s filter expression parameters[1], which is required for the Load Balancing Process. It is important to recall that the PNs must update their filters to start/stop receiving data items assigned to some *Slices*. To work around this CoreDX DDS limitation the filtering was implemented within the PN code, instead of the DDS layer. The drawback of this workaround is a higher network utilization: all data items are delivered to all PNs , but all the data items that are from *Slices* not assigned to the PN are discarded. The advantage of this implementation strategy is that the current DPSLB prototype can be used also for applications where data processing depends correlating data items of different *Slices,* e.g. in order to process a data item *A* the PN needs a data item *B* assigned to another Slice than A.

The local cache for each *Slice* is stored as a map indexed by the *Slice* ID. Thus, each entry in this mapping stores a collection of data items assigned to the same *Slice*. To avoid misinterpretations, throughout the next sections this mapping will be called local cache, and a single entry will be called local *Slice* cache.

1 Apparently this is a CoreDX DDS bug, since the OMG DDS standard determines such capability.

```
public interface ApplicationDataReaderListener<TopicType> {
  public void onNewData(TopicType topicSample);

  public void onNewCorrelatingData(TopicType topicSample);
}
```

Figure 18 – Application listener interface

To inform the application when some delivered data item should be processed – and when it should not – the PN has two callbacks to the application, one to inform a normal data item data must be processed – named *onNewData()* – and another to inform a data item that would be discarded by the PN but may be useful for the application for data correlation – named *onNewCorrelatingData()*. Figure 18 shows the listener interface that is used by the PN layer to notify the application. Both methods, *onNewData()* and *onNewCorrelatingData()*, have a parameter named *topicSample* that contains the data item received. The *ApplicationDataReaderListener* uses the technique of Java Generics [53] [54], allowing the *ApplicationDataReaderListener* to manipulate objects of various types while providing compile-time type safety and to eliminate the drudgery of casting [55].

```
public Object createTopic(String typeName, String topicName,
  boolean balanceTopic);

public Object createContentFilteredTopic(String typeName,
  String topicName, Object topic, String logicalExpression,
  List<String> parameters, boolean balanceTopic);

public <TopicType> boolean createDataReader(Object topicDescription,
  ApplicationDataReaderListener<TopicType> applicationDataReaderListener);
```

Figure 19 – Methods to create a Topic, Content Filtered Topic and Data Reader

Three most important methods at PN, shown in Figure 19, are: *createTopic()*, *createContentFilteredTopic()* and *createDataReader()*. The *createTopic()* creates a DDS Topic of the class *typeName* (e.g. “br.puc.rio.inf.lac.FooTopic”) with the name *topicName* (e.g. “My Example Topic”), and where parameter *balanceTopic* determines if this topic should, or should not, be balanced by the DPSLB solution. Apart from *typeName*, *topicName* and *balanceTopic* parameters, that are the same as with *createTopic()* method, the *createContentFilteredTopic()* creates a Content Filtered Topic from an original *topic* using the filter expression *logicalExpression* and the filter expression´s parameters named *parameters*. The last method, *createDataReader()*, subscribes the PN to the Topic *topicDescription* using the *applicationDataReaderListener* as the listener that the PN uses to notify

the application. In order to correctly notify the application, PNs hold a mapping that stores all *Application Data Reader Listeners* indexed by theirs Topic´s name.

After creating the Data Reader (i.e. a subscription to a Topic), the PN is able to receive and notify the application about data items. The DPSLB solution is completely transparent to the application developer as he/she has no further configurations to do. In fact, he/she only needs to inform whether the topic should be balanced by the DPSLB, or not.

When a PN receives a data item (or *sample* in DDS jargon), PN checks whether the data item is assigned to a valid *Slice*. In other words, it checks whether it is responsible for processing data items for this *Slice* and then, in case of a valid *Slice*, the PN notifies the application through *onNewData()* method. Otherwise, PN checks whether the *Slice* is *In Load Balancing Session* state, which means that data items of this *Slice* are to be cached because the PN is involved in a *Load Balancing Process*. Finally, if the *Slice* is not in *Load Balancing Process* it means that this data item should not be processed by the application, in which case PN notifies the application about a data item that may be discarded through *onNewCorrelatingData()*.

4.3.1 MS and LES

As aforesaid, PNs utilizes MS to periodically monitor its resources and it is configured to check its CPU and Memory usage every two seconds, but this parameter can be easily modified. MAPE-SDDL is able to monitor other resources such as Disk free space or Swap Memory utilization. However, for the sake of simplicity, in the DPSLB prototype only CPU and Memory monitors were deployed.

LES is configured to notify the APS in the Load Balancer every time that occurs any change on the monitored resources. Even though a more complex utilization of MAPE-SDDL is possible, the main goal in this work was simply to validate the proposed DPSLB solution, instead of developing sophisticated monitoring and notification mechanisms.

4.3.2
CES and Load Balancing Process

As mentioned in section 4.3, a PN executes the *Executing Service* of the CES while the *Control Service* is executed by the Load Balancer. CES is the adaptation engine that enables PNs to receive Actions for moving Slices as a consequence of a load redistribution *ActionPlan* produced by the *Control Service*. The Executing Service is the *Effector* in the MAPE-K loop, described in section 2.4.1. The Actions supported by PN are: *addSlice*, *removeSlice*, *updateSliceState* and *sendCacheToNode*. Figure 20 illustrates the possible transitions between the *Slice* states on each PN and their interaction with the Action. The *Slice* state *Not In Use* means that data items assigned to this can be discarded by the PN because another PN is processing these data items. But due to the CoreDX DDS´ limitation, mentioned in section 4.3, each PN has access to all *Slices*. Therefore the *addSlice* and *removeSlice* Actions do not actually add or remove a *Slice*, but only change the *Slice* state.

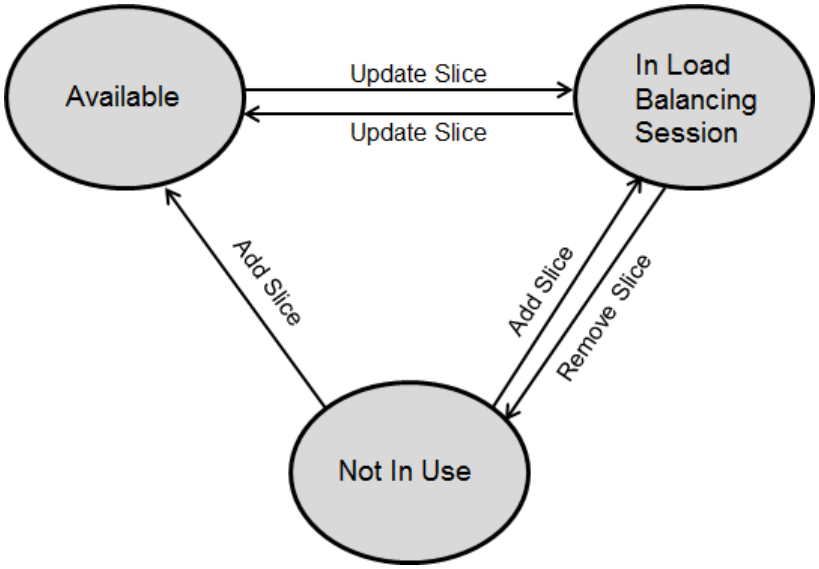

Figure 20 – Slice transitions on a PN

The *addSlice* Action has two parameters, the *Slice* State and a collection of *Slices* to be added. This Action changes the *Slice* state from *Not In Use* either to state *Available* or to state *In Load Balancing Session*. The first PN that shows up in the system has its *Slice* states changed from *Not In Use* to *Available* by the Load Balancer since there is no other PN to start a *Load Balancing Process.* The following PNs that shows up receives by the Load Balancer a *updateSliceState*

informing that it has to change the *Slice* states involved in the Load Balancing Process to *In Load Balancing Session*.

The *removeSlice* action, accordingly, informs that a PN should stop processing data items for the specified *Slice*. This Action has only one parameter, which is a collection of *Slices* that must be removed. This Action is sent to the *Slice-giving* PN after the *Slice-taking* Processing adds its *Slice*. It is not a valid transition to switch from *Available* state to *Not In Use* state since a *Slice* needs to be securely moved to another PN while the *Slice* is in the Load Balancing Session state.

In Load Balancing Process, both *Slice-giving* and *Slice-taking* PNs should update the state of the involved *Slices* to *In Load Balancing Session*. The *updateSliceState* Action has a parameter that informs the new *Slice* state and another that is a collection of *Slice* that will be updated. After the *Slice-giving* PN updates and removes the *Slices*, the *Slice-taking* PN can proceed with the update action.

Finally, the *sendCacheToNode* is the Action that sends the local *Slice* cache from *Slice-giving* to *Slice-taking* PN. This Action has two parameters: the *Slice-taking* PN ID and the set of *Slice* local cache that are to be sent. After the send operation, the *Slice-giving* PN also cleans its local cache that stored the data items assigned to the specified *Slice*. When the *Slice-taking* receives the local cache from *Slice-giving*, it merges its local cache with the remote cache received, as described in section 3.4.

```
struct CacheTopic
{
    DDS_KEY long sliceId;
    long long leastSignificantBitsSenderId;
    long long mostSignificantBitsSenderId;
    long long leastSignificantBitsReceiverId;
    long long mostSignificantBitsReceiverId;
    sequence<octet> dataItemCollection;
};
```

Figure 21 – Interface Description Language (IDL) of the CacheTopic

The data items of a *Slice* cache are sent through a DDS Topic named *CacheTopic*, illustrated in Figure 21. This IDL can be compiled for many programming languages. After compiled, CoreDX DDS generates the Java Class shown in Figure 22. The *CacheTopic* carries fields to inform the *Slice ID* (*sliceId*), *Slice-giving* PN ID (*leastSignificantBitsSenderId* and *mostSignificantBitsSenderId*),

Slice-taking PN ID (*leastSignificantBitsReceiverId* and *mostSignificantBitsReceiverId*) and the data items (*dataItemCollection*). The *Slice-giving* and *Slice-taking* IDs are each composed of a Java UUID (Immutable Universally Unique Identifier). This UUID represents a 128-bit value that is separated into two Java *long* variables to be sent through the DDS Domain. The collection carrying the data items is serialized into a *byte* array.

```
public class CacheTopic {
  public int    sliceId;
  public long   leastSignificantBitsSenderId;
  public long   mostSignificantBitsSenderId;
  public long   leastSignificantBitsReceiverId;
  public long   mostSignificantBitsReceiverId;
  public byte[] dataItemCollection;
};
```

Figure 22 – Generated CacheTopic Java Class

The *Slice-taking* PN, after receiving a *CacheTopic sample*, deserializes the *dataItemCollection* and gets its *Slice* local cache. With both local and remote caches, the *Slice-taking* PN uses a Java *Set* in order to generate the Merged Cache, which is a set that has no duplicated data items. With the aim of uniquely identifying each data item, it is mandatory that all data items hold an ID field, that can be a Java *int* or *long*. This ID is utilized by the Java Set to avoid the insertion of a data item already added in the Merged Cache. After generating the Merged Cache, the *Slice-taking* PN removes its local *Slice* cache and delivers the data items to the application through their *Application Data Reader Listeners*.

The prototype implementation does not transfer any application state of the PNs associated with the moved Slices, but only the data items that were cached. Such state may be a history and statistics about the processing data items that were generated by the application. However, the lack of such state transfer is not a limitation of the DPSLB approach *per se*, but has been avoided only in the proof of concept prototype, for the sake of simplicity.

4.4 Load Balancing Algorithms

In order to validate the proposed solution, one load balancing algorithm – which implements the *LoadBalancingAlgorithm* interface – was developed to be used by the Load Balancer. Since the *Load Balancing Algorithm* needs not take

into account how to monitor the PNs, generate and control the ActionPlan execution and manage a data structure that stores the PN information and their Monitoring Event Notifications, its development quite easy.

Since the goal was only a proof of concept, the implemented algorithm is quite simple. The idea is to assign always more/less the same number of *Slices* to each PN. To do so, the algorithm calculates the average number of *Slices* per PNs and then classifies each PN in one of two groups: PNs that have more *Slices* that the average (*Slice-giving* PNs) and PNs that have less *Slices* that the average (*Slice-taking* PNs). If there is a single PN, all *Slices* are assigned to it, otherwise the algorithm gets the first *Slice-taking* PN and *Slice-giving* PN and checks how many *Slices* can be moved to the *Slice-taker*, thus generates and adds on its own list of *Slices* to move that informs *Slices* to be moved from *Slice-giving* to *Slice-taking* PN. After this step, the *Slice-giving/taking* PN that has no more *Slice* to be moved/received is removed from its list. When there is no more *Slice-giving/taking* PN, the algorithm returns the collection of *Slice Movements* to the Load Balancer, which in turn will generate the ActionPlan and send the Actions to the corresponding PNs.

In a scenario where there are two PNs, *A* and *B*, and ten *Slices* (*A* and *B* have five *Slices* each one), the following steps would be executed if a new PN C showed up:

- Calculate the average number of *Slices* for each PN ($10 / 3 \approx 3$);
- Classify the PNs in *Slice-giving* PNs [*A*, *B*] and *Slice-taking* PNs [*C*];
- Check that *A* can give two *Slices* ($5 – 3 = 2$), thus move two *Slices* to *C*;
- Remove *A* from *Slice-giving* list [B];
- Check that *B* can give two *Slices* ($5 – 3 = 2$), thus move two *Slices* to *C*;
- Remove *B* from *Slice-giving* list [ø] and *C* from *Slice-taking* list [ø];
- Return two Slice Movements, one to move two *Slices* from *A* to *C* and another to move two *Slices* from *B* to *C*.

In the given example, *A* and *B* would be assigned to three *Slices* and *C* to four *Slices*.