Service-oriented Operating Systems

First Implementation Set: Protocols

Project Deliverable D3.2

Bruno Santos, Javad Zarrin, João Paulo Barraca, Rui Aguiar [IT]
Christiaan Baaij, Jan Kuper [UT]
Dario Faggioli, Giuseppe Lipari, Tommaso Cucinotta [SSSA]
Aleksandar Dragojevic, Vasileios Trigonakis, Vincent Gramoli [EPFL]
Daniel Rubio Bonilla, Lutz Schubert [USTUTT-HLRS]

Due date: 30/04/2011
Delivery date: 30/04/2011

SEVENTH FRAMEWORK PROGRAMME

This work is partially funded by the European Commission under
FP7-ICT-2009.8.1, GA no. 248465

Version History

Version	Date	Change	Author
0.1	09/03/11	First TOC	Javad, Bruno
0.2	19/03/11	Initial Contributions	All
0.3	21/04/11	Final Contributions	All
1.0	30/04/11	Document Finalisation	Bruno, Javad

Table of Contents

1. List of Figures

2. List of Tables

3. Abbreviations

Abbreviation	Description
S(o)OS	Service-oriented Operating System(s)
IDL	Interface Definition Language
NIC	Network Interface Card
RPC	Remote Procedure Call
SDP	Service Discovery Protocol
SOA	Service-Oriented Architecture
Functor	A C++ Function Object
SRDP	Service/Resource Discovery Protocol

I. Introduction

WP3 concentrated on communication models and protocols in distributed operating systems on future many-core systems.

At the base of any distributed system lie a set of protocols making it possible for entities to exchange data. These protocols must support the interaction models of the upper layers, as well as provide the correct primitives for optimal operation. If the primitives provided are not adequate to the environment or to the services transacted, the result will be a deficient operation of the entire system. The impact of data encoding, communication model and primitives provided can be a major obstacle for efficient distribution of kernel functions over an loosely coupled, heterogeneous environmental. Even if primitives are adequate, data coding methods must be designed so that latency is minimized while overall impact in the system is kept under strict values.

The challenge of these environments is to devised communication solutions capable of enabling a distributed kernel operating over a environment with more natural unpredictability. Latency guarantees or bandwidth are not within known bounds but can present value covering more than 3 magnitudes. Also, message passing solutions, which are typically proposed by distributed environments, must cope with concurrency over links with different latency, and even message loss due to hardware failure, link failure or link congestion, while providing scheduling assurances and fairness.

Like in the remaining of the S(o)OS work packages, the work and ideas discussed here consider an highly heterogeneous environment, where computational resources (CPU, GP-GPUS, etc.) can have much diverse characteristics and can be located either in the same dye, same board or across an high bandwidth link.

1. Document Organization

In chapter II we propose a kernel communication framework based on the component based architecture. In this model, components are opaque, exposing only the interface to offer its services. An opaque component enables heterogeneous and message passing solutions, thus components can be migrated and distributed.

In chapter III we propose a resource discovery protocol for large many core system. In order to benefit from all the potential resources in a large many core system, communication protocols are required to recognize and locate resources with deep understanding of the resource capabilities according to a query requirements.

In chapter IV we present tools for network modelling and simulation, in connection with the work in other work packages. This work is also connected to the resource description in chapter III.

In chapter V we present protocols, in connection with the work in other work packages, for message-passing and replication. This work is also connected to the component kernel in chapter II.

II. Component based Kernel

We propose a component based approach to realize the modular and distributed communication model for future many-core and heterogeneous systems.

1. Overview

A component or service has a clearly defined interface and conforms to an expected behavior. Components communicate with each other through their interfaces. The modularization is achieved because the implementation is completely opaque to the user thus it can be replaced or upgraded, it can also have multiple and/or heterogeneous implementations. The system can decide which implementation to plug at runtime.

Component based architectures are not new, they have been traditionally used to implement language neutral interfaces using an IDL[1], RPC[2] and allow easy replacing or updating existing components without breaking the system. Examples of such are COM[3], XPCOM[4], CORBA[5].

A component can interface with another component whose implementation delegates the call, using message passing, to another node, thus enabling a distributed and/or heterogeneous system. This also greatly simplifies the programming of such system, since the programmer doesn't have to explicitly use message passing.

One the requirements for the proposed approach is to be able to replace at runtime the implementation of a given component, thus enabling migration to another core while maintaining a functional and correct interface at the original node. This is possible at the kernel level, since it has control of the execution and interruptibility of cores.

Components

A component in kernel are the objects that represent a thread, process, a file or even an hardware device. Some of these can be physical bound, such in the case of hardware devices, or migratory, such as a thread or process.

Comparison with “service-oriented”

In service-oriented architectures the client-server model is enforced, this is the main difference, but for a kernel it would be inefficient to have a pure client-server model and a component provides a service.

Migration and Replication

Components can be migrated between cores and/or distributed among cores. When a component is not locally available, a stub implementation is used locally instead. The stub delegates the request through message passing to the current object location. However, a component location may change and on a many core system it would no be unfeasible to keep track of the location of component.

To make addressing location independent, each component instance has a GUID. To resolve a GUID to physical location we take the consideration that migration of components is a rare event, thus a sensible approach is to cache the last known physical location. If the location changes, an error

1 http://www.omg.org/gettingstarted/omg_idl.htm
2 http://en.wikipedia.org/wiki/Remote_procedure_call
3 http://www.microsoft.com/com/
4 https://developer.mozilla.org/en/XPCOM
5 http://www.omg.org/spec/CORBA/

indicating this is sent as a reply when a message is sent to the previous expected physical location. To locate a resource that changed location, a discovery procedure of the component must be performed.

Managing Component Lifetime

Another challenge to a distributed system is managing the lifetime of a component. Each core is responsible for managing the creation, lifetime and deletion of each component. Typically, a reference count is used to maintain lifetime. However, on distributed system, this approach will need to take into account the potential unreliable state of other cores. The sensible approach is to keep separate reference count for local and external references. If local reference counting is zero and faults have been detected on the system or a sensible amount of time as passed since the component was last accessed and based on system load, an enumeration of the given component can done to detect dead references.

Interface Description Language

To facilitate code development, an interface description language is used to describe a components interface, from this code can be automatically generated for:

- message passing interface and handling.
- headers for a C/C++ implementation.

Message Passing

Each core must handle messages received from other requesting cores and dispatch them to respective component. At the core handling facility, there are stub servers, for the components located on this core, that handle the respective messages. The core handling facility can also hold on queue messages to a given component instance and route the messages to another core, in order to support migration.

2. Protocol

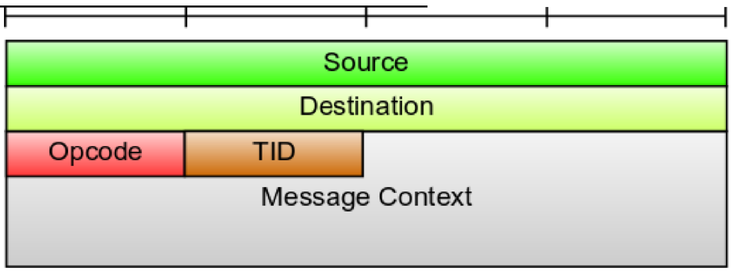


Figure 1: Protocol for Component Message Passing

The protocol depicted in Figure 1 was developed for RPC, migration, replication and management of components. The *source* and *destination* are the GUIDs of the component instances of the originator and recipient of the operation identified by the *opcode* field. The remaining of the message is specific to the operation and dependent on the component.

A range of values for the *opcode* field is reserved for common operations and protocol internal operation, the remaining is for the component specific use, that is typically to identify which operation/service is being requested. The *TID* field is the transaction identifier, for distinguishing individual operations since several operations can be issued from a single core almost simultaneously.

For communication with remote components, the client stub[6] caches the last known good physical location, thus a message is always sent to last known physical location. Each core has a table that keeps tracks of all existing component instances that have remote stub references. Upon receiving a message, a lookup is performed to get the recipient instance and dispatched using the respective service stub[7]. If the lookup fails, then a message reply is sent indicating that the instance does not exist, upon receiving the reply the originator must perform a lookup using the mechanisms for resource discovery, see chapter III. 2.

To manage the lifetime of component instances, for each instance, a reference count is keep for local and remote references. When a component is migrated, first the local references are replaced with remote references, a then the component can be migrated with a local reference count of zero. Remote references only track on a core basis, that is, if core A has 4 references to a component instance in core B, then core B only counts 1 remote reference. This minimizes the amount of messages needed to exchange, thus core A only needs to notify core B when its has 0 references to a given component instance in core A. The client side keeps track of the local references, of course, no remote references in this case are tracked since that is the exclusive responsibility of the server side.

During a migration operation, any messages receive whose recipient is the component instance being migrated are queued and then dispatched to proper destination when the migration is complete.

3. Distributed Operation

In a distributed OS there's is a number of system which are global to the entire systems. This systems must be fault tolerant, thus requiring replication and a consistent view along the entire system.

One these key systems is the global namespace. Some examples of what a global namespace can provide are:

- access to files
- access to devices such as printers
- connectivity with IPC

The namespace aggregates a number of heterogeneous resources into a single file system like view. A namespace does to kernel resources what a DNS system does to web pages, provides a means to a access these resources without knowing their location.

A distributed namespace is also key system for distributed file systems, thus enabling distributed storage and replication for a fault tolerant and efficient system.

Designing such system will require communication protocols that guarantee a consistent view among replicas and atomicity in certain operations, such as: create; rename; delete. For instance, the Barrelfish OS uses a two-phase commit (2PC) synchronization protocol that relies on multicast to guarantee synchronization among distributed replicas. Critical data is stored at each core and it must be kept consistent among the replicas, so the cores communicate through messages in order to execute a 2PC agreement protocol. This research topic is further explored in chapter V. 2.

6 The client stubs creates, serializes and requests the message to be sent to the recipient.

7 The service stub is responsible to de-serialize the message and call the respective method.

4. Design and Implementation

One of the greatest challenge for many-core communication is the asynchronous nature of message passing. To overcome this challenge on a synchronous paradigm, it would require using multiple threads. However, threads are an expensive resource and with the additional overhead imposed by context switching makes this a non scalable solution. On the other hand, the state of the art shown us[8] [1][2] that a non-blocking communication paradigm, such as the reactor [1] and proactor [2] patterns, provides a scalable solution to this challenge.

We are developing a novel kernel from the ground-up inspired on this principles, to advance and investigate new methods for service oriented, heterogeneous and loosely coupled distributed operating systems for the many-core era.

The research kernel being developed is written in the C++11 programming language, the rational for this choice was the low level systems programming support combined with high level constructs such as template meta-programming and lambda expressions which greatly simplifies asynchronous programming.

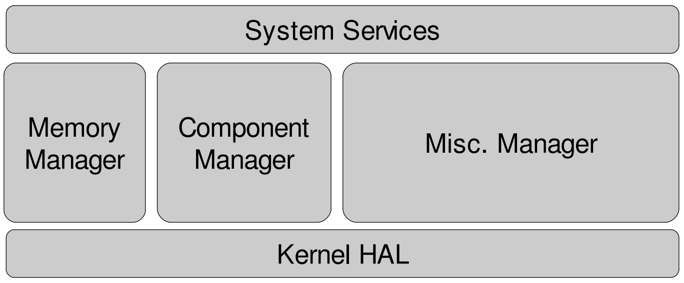


Figure 2: Kernel Architecture: abstraction layers

Figure 2 shows the kernel architecture into 3 layers of abstraction. The 1st layer, the kernel hardware abstraction layer, compromises the interrupt handling, scheduling and related primitives, exception dispatching, system services dispatching and the memory management unit abstraction.

At the 2nd layer we have the memory manager, the component manager and other possible management duties for components, such as threads and processes, which may also be implemented at the user-level like in a micro-kernel.

At 3rd layer, kernel services are exposed to user-mode. This layer is more of a security boundary for proper validation of parameters and permissions for user-mode system service requests.

a) Asynchronous Support

To perform an asynchronous operations, in terms of software, a callback function is provided by the caller to the called method. Upon the completion of the operation the callback function is invoked with the result of failure or success of the operation. When the called method returns, the operation may or may not have been completed, thus the result is always reported to the callback asynchronously.

In a pure C programming language style, programming complexity may arise when the programer needs to pass additional information to the callback. In the mentioned style, the common approach is to supply to the called method an additional pointer argument which will then be supplied to the callback. If the additional information does not fit in the pointer argument, then the programmer must create a structure to hold the additional information, allocate memory for it and pass the address in the pointer argument and then free the memory in the callback. This added complexity is due to the lack of language features that can simplify this. Since the kernel is written in C++11

8 More information available at http://msdn.microsoft.com/en-us/magazine/cc302334.aspx and http://www.kegel.com/c10k.html.

programming language, functors are the adopted callback mechanism which can hold additional information with much simpler interface, using the lambda[9] syntax, compared with the C style approach. For wording purposes, this callback mechanism is also designated as a completion functor. The code listing below illustrates an asynchronous method call for a read operation:

```
read(buffer, [buffer](error_code error, size_t transfered) {
   //thinking asynchronously: sequential execution resumes
here
 });
```

Calling an asynchronous method

In the illustrated asynchronous method call, the *buffer* argument is a tuple with the address and size of the memory region to hold the data to be read, the second argument is the completion functor expressed using lambda syntax. In the lambda the *buffer* argument is captured by the lambda capture list denoted in square brackets, the arguments *error* and *transferred* are passed when the completion functor is invoked indicating respectively the success or failure of the operation and the number of bytes transferred.

A new type of primitive was developed to meet the requirement of the adopted asynchronous mechanism. This primitive, known as ***async***, provides a stack based mechanism to store the completion functors. When a completion functor is pushed onto the stack, a reference to a tuple is returned to store the results to be passed to the completion functor. The ***async*** primitive can be waited on, where a completion functor must also be provided for this. When a wait is performed no more completion functors can be pushed onto the primitive until its signaled, causing the wait to be satisfied, and trigger the sequential execution of the completion handlers until either the last one or a new wait is performed. Also, only an error code can be passed to the wait completion functor, which is the argument supplied to the signaling method.

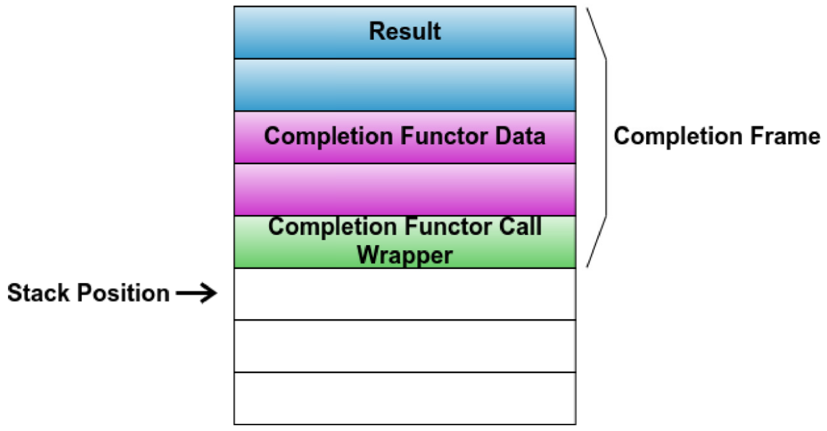


Figure 3: Async primitive stack with a pushed completion functor

Figure 3 illustrates the context of the asynchronous stack when a completion functor is pushed. Space is allocated[10] for the result data and initialized to its default values. Then space is allocated to the functor data and copied from the supplied functor. Finally, a pointer to a wrapper function is pushed at the end. The wrapper function unwinds the stack, and invokes the functor with the result data as arguments. This is accomplished through the C++ meta-programming features.

9 In C++ lambdas are "syntactic sugar" for function objects.

10 Space is allocated by manipulating the stack position pointer. The current implementation uses a fixed stack size. And, like in a thread stack, an overflow will result in a kernel panic.

```
 template<class Completion>
 void read(const tuple<void*, size_t>& buffer, Completion
comp)
 {
   auto astack = ke::get_current_async();
   auto result = astack.push<error_code, size_t>(comp);

   if (do_read(buffer, result))
      astack.pop();
 }
```

Implementing an asynchronous method

An asynchronous method first requires an entry template wrapper, as illustrated in the code listing above, in order to set the completion frame in the current **async** primitive. Then it calls the method that implements the operation. The operation may complete immediately. In this case, the completion frame must be popped for the completion functor execution.

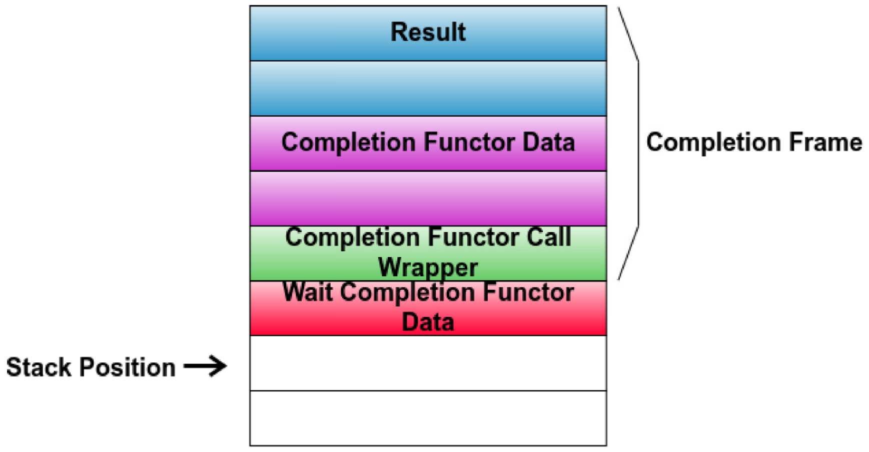


Figure 4: Async primitive stack with a pending wait

Figure 4 illustrates the case when a wait is performed on the **async** primitive. Considering the previous code examples, if a read operation had to issue an hardware I/O request, it would first set the **async** primitive into a wait state, then it would queue necessary information such as the reference to the **async** primitive. Next, it would issue the request and return indicating a pending operation. Later, when hardware interrupt fires, the **async** primitive is signaled to notify the completion of the operation.

When a wait is performed on the **async** primitive, it is moved from being the current one to the **async** primitive wait list and a new **async** primitive will be allocated from the core pool when a new asynchronous operations is initiated. The pointer to the function wrapper for the wait completion functor is stored on dedicated data member of the **async** primitive. This allows the interruption of signaling process when on of the completion functors performs a wait.

The **async** primitive is bounded to the core[11] where the operation is initiated. It arbitrary executes in any thread context[12] when it gets signaled, thus avoiding context switching overhead and cache thrashing.

The **async** primitive can also be bound to a thread, this is particular relevant when servicing a request on behalf of user-mode and access to its address space is required.

11 Depending on the architecture, this could also apply to small group of cores that share a cache.
12 There are limitations imposed by a core interrupt level, thus at certain interrupt states the execution can be deferred until the interrupt level drops.

III. Resource Discovery and Matching

Service and Resource Discovery (SRD) is the process of finding and locating all the available Services/Resources within the network. Today, with the growth of size and diversity of the computer networks and architectures, SRD solutions are becoming more enhanced and widely spread for different applications, environments and infrastructures. There are various approaches possible to perform resource discovery which are differentiated from one another in the following aspects.

Architecture: SRD architectures are Structured (Directory Based), Unstructured (Directory Less) and Hybrid:

- **Structured:** majority of the service discovery (SD) systems are using a structured architecture. By structured architecture we can understand systems on certain nodes of which the services information is fixed, such systems are in general faster in discovering services than other SDAs, with a fixed and predictable time of service discovery. Structured SDA are classified into Centralized and Distributed ones. In centralized or directory system, the information about the services which are provided by all other nodes is stored on preconfigured or auto configured nodes in limited number. In this system nodes or the directory, periodically update the registered service information in central repository. But some time these services are not valid or expired because they are not updated.
  In centralized architecture, we have typically the fastest search time, less traffic and overhead, high rate of service discovery . But potentially this system has some points of failures such as high rate of false service discovery (discovering invalid or expired services) ,and being vulnerable to Denial of Service (DoS) attacks.
  In decentralized systems, the distribution of service information is controlled by SD system and the information is almost distributed between all the participant nodes in a predefined manner. Overlays of this type commonly make use of Distributed Hash Tables (DHT). Decentralized systems can be extremely effective for searching resources using unique identification, but they fail when a search using partial marching is required. Another drawback is creating additional overhead by managing network architecture (by updating the system structure in the case of node failures or node arrival).
- **Unstructured:** distribution of the service information among the nodes is not followed by a predefined or controlled mechanism. In spite of decentralized structured system, the system built as unstructured or classical peer to peer (UP2P) supports partial matching and also complex querying. The use of loose architecture makes the system resistant to node failure and DoS attacks; another advantage of this architecture is easy adaptation of the system to frequent node joins and disjoints. On the other hand, UP2P has low-level rate of service discovery in comparison to the structured systems. Nodes in these system are free to behave as they want , however they carry a part of the network functionalities and their failure or misbehavior can be costly .There are some other challenges with this kind of SD systems specially related to issues such as fault tolerance under churn, load balancing ,and flash crowds.
- **Hybrid:** The classical P2P service discovery architecture do not scale well in the large networks , that is why it should be replaced with the peer to peer overlays that utilize a hybrid architecture. The hybrid solution tries to combine the advantages offered by the classical structured and unstructured service discovery systems.

- **OSI layer positioning:** SDPs can be built based on different layers within protocol stacks but most of the SDPs rely on the network layer or higher application layers such as Service Location Protocol (SLP), Universal Plug and Play (UPnP) and Simple Service Discovery Protocol (SSDP), however, to increase total efficiency and performance of the system in mobile ad-hoc networks SDPs are integrated with the routing protocols in the network layer. For the fixed network, the appropriate solution is implementing service discovery in system or application level.

Service Announcement: SRD Mechanisms for service distribution are Pull (Reactive, Query-based) and Push (Proactive, Announcement-based) and Hybrid. Reactive is the mechanism in which a node sends an explicit request query to find and use a specified service, and proactive is a state that the node waits to receive a service announcement which is advertised by other service providers. Proactive uses the periodical service broadcasting or multicasting, and creates more traffic and this is why today majority of SRD frameworks are using reactive or hybrid methods.

Network Type: the SRD solutions are different due to the type of the network infrastructure (fixed, wireless, constrained resource, mobile or sensor networks) for which they are defined . Some of SRD methods may support different type of network but they are not always efficient.

Scale: Scalability of a SRD is generally related to its delimitated scope and provides issues on suitability and consistency. Consistency in service discovery protocol means that the discovered services must be matched with the actual services which are advertise by the origin nodes. While the size of a network becomes larger, the number of service entities also increases, however, the network parameters such as packet loss, link disconnection, and transmission delay is almost the same and therefore the inconsistency or rate of false service discovery between the original services and discovered services is raised. The mechanisms which are used by SRDs to reduce inconsistency regularly poll the service status or subscribe to the service status modification from the clients sides and produce service change notification from the server sides. In centralized architecture especially with the push service advertisement method, the status of services periodically are being updated which causes a noticeable increasing in the number of expired or invalid services compared to the distributed architectures and pull based Sds.

Scope: SDPs generate discovery messages over the network and thus it is important to limit distribution of unnecessary messages in the environment by defining a scope for service discovery. Scope enhances service discovery solutions by avoiding considerable amount of unneeded processing on client or server nodes. Definition of service discovery scope is made according to network topology, context information or user rules.

Routing Protocols: Service Discovery Protocols can be either pure discovery protocols or coupled with routing protocols which address the optimum path from the origin node to the destination node, such as Ad hoc On Demand Distance Vector Protocol (AODV), Dynamic Source Routing (DSR) (reactive) , On-Demand Multicast Routing Protocol (ODMRP), and Optimized Link State Routing Protocol (OLSR) (proactive).In wireless network most of the SRDs are being included containing their particular routing protocols or are built based on an existing routing approach.

The resource discovery which is required for the So(o)S project is going to advertise resources (processors and cores in multi-core nodes) as services according to Service Oriented Architecture (SOA).

Consistency: Consistency in service discovery protocol means the discovered services must be matched with the actual services which are advertise by the origin nodes, while the size of a network becomes larger, the number of service entities also increases, however, the network parameters such as packet loss, link disconnection, and transmission delay almost is same, therefore, the inconsistency or rate of false service discovery between the original services and discovered services is raised.

The mechanisms which are used by SRDs to reduce inconsistency are regular polling the service status or subscribing for the service status modification from the clients sides and producing service change notification from the server sides. In centralized architecture especially with the push service advertisement method, the status of services periodically are being updated, it causes a noticeable increasing on the number of expired or invalid services in compare with the distributed architectures and pull based SDs.

1. Multi-Layer Resource Description

Multi-layers resource description model is designed for the purpose of resource discovery to provide required information about resources for the queries in easy and efficient manner. It categories resource specifications includes computational and communicational properties and behaviors in the there layers from more abstract information to more detailed characteristics. In resource discovery we are considering to avoid generating overload for discovery traffic. By using Multi-Layer resource description model we just transmit necessary information and resource description according to the status of the discovery procedure. Descriptions in Layers are classified according to the following categorization:

Layer1: Inter-Cores Level (Network on Chip (NoC) Level), resource description in this level has been characterized according to THE type of information which is directly related to the core specifications. The most typical information in this level is floating point and integer operation per cycle and private cache size.

Layer2: Inter -Chips Level, describes the specifications of the edges for the core communications, interconnection networks and memory hierarchy which include information such as: bandwidth, latency, shared or global cache size and number of cores.

Layer3: Inter -Nodes Level, Resource description covers a total overview of all the communications, memory and processing features and aspects in the node such as memory size and number of processors.

In the S(o)OS deliverable “D2.2 First Implementation Set: Hardware” [3] we have discussed more about the detail of multi-layers resource description model.

2. Querying and Search Algorithms

Search techniques are the main part of SRDs. There exist several various techniques to locate resources and services in the network. For small network with limited number of resources no complex search method are required, simply a node can discover other resources by using basic broadcasting or multicasting. Directory-based or centralized systems with limited number of servers also do not require a complex propagation method for querying, however, in distributed networks, such as unstructured peer to peer overlays, to support complex or free-form queries, appropriate search techniques have to be applied and integrated with the query propagation methods to increase efficiency and scalability. The search algorithms are categorized in two groups, informed and uninformed search methods:

- **Uninformed:** in uniformed methods, the sender nodes know nothing about other nodes and resources in surrounding network . These methods can be either systematic or random algorithms, in systematic approaches, to find a resource, almost whole or part of the search tree or graph is explored, and there is no option for random or probabilistic search in systematic manner. The most well known search algorithms in this group are Breadth First

Search (BFS) and Depth First Search (DFS) .However there are several other considerable systematic and random search methods such as Flooding, Depth Limited Search, Iterative Deepening, Uniform Cost Search, Random Walk, etc. According to the aforementioned search methods, there are various SRD protocols which utilize and integrate them with their query propagation methods such as Probabilistic Flooding and Forwarding Protocol, Gossip-based Protocol, etc.

- **Unicast and Multicast:** Unicast is the most conventional communication way in SDPs. The origin node (sender) explicitly addresses the receiver and sends messages (query, reply, advertisement, etc.) directly to the destination node through network. This is the most efficient way of the communication while the receiver's address is clear. However in pervasive environment with wireless and mobile ,node services and even nodes are not known in advance and therefore we need to use, (in this approach), another alternative which is multicasting number of nodes' starts to establish a multicast group by sending unicast initiative messages to each other. Once a node sends a message to a multicast address it causes several unicast messages from the sender to all the members of the multicast group. Multicast is used when the destination nodes are unknown and it is not clear which nodes are offering a particular requested service, though this communication model compared to broadcast is still efficient.
- **Broadcasting:** or simple flooding, is using one to all communication to send packet for all the nodes in the network or subnet .Service discovery systems employ broadcasting in the absence of the predefined servers which creates a lot of overhead in the network, and generally it is used to advertise new services in the wireless network which are more adapted to the nature of the broadcast communications. Today broadcasting is mainly used in the small wireless network with limited number of services.
- **Publish/Subscribe:** It is similar to multicasting in nature. In this approach, a receiver node subscribes for certain services which are offered by publisher nodes and obtain these services directly or through intermediate nodes. Publishers constantly report all the service changes to their subscribers. They may update intermediate nodes. Subscribers are not updated immediately and they should fetch new data from the middle nodes as soon as they contact them, instead of direct updating.

Algorithm 1 demonstrates steps of resource discovery in a large cluster with multi-CPUs and multi-core nodes. We have developed a resource discovery protocol using MPI to discover all the available processing resources in the network for a particular query. To implement a resource discovery procedure the following components have been defined accordingly.

- **Discovery Event:** This is an event of discovery procedure which is assigned to all the processing resources in the network. It is triggered when the correspondence resource has overload instructions. Next, it is required to distribute the overload processing tasks over other available resources. Discovery Event initializes the procedure to discover requested resources.
- **QMS:** Queue Management Systems are the components which reside in each node or CPU. They are responsible for checking the availability of their processing units and manage the queue of individual queries from various origin cores.
- **DHT Resource Mapping:** Distributed Hash Tables are located in resources to keep the state of execution by mapping query/job/instruction IDs with the most high-ranking discovered resources.

- **RCT:** Resource Cost Table is a data structure to record the rank of discovered resources according to resource ranking algorithms which can be used for future resource discovery. There is a time validation for RCT after which the RCT would be invalid.

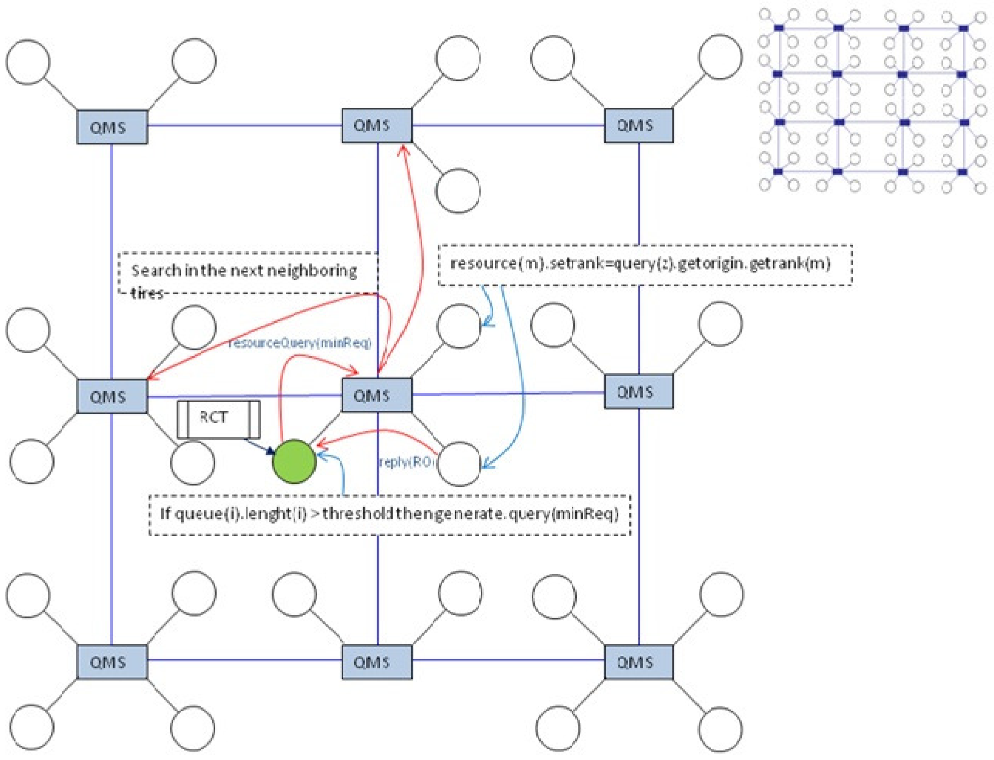
```
 Data: query = SetInitialArguments
 Event:  discoveryEvent = SetTrigger
 Data: timing = Calculate timing
 Data: RCT//resource cost table
 Data: QMS, remoteQMS //queue management system
 Data: reqMessage, repMessage
 Collection: manyCoreSystem=Set{n, f(n_id)} // n nodes, f(n_id) cores per
node
 Collection: multiCastGroup, qualifiedResources , remoteQMS_group,
DHT_resource_mapping
 Set timing = false

 /* Main discovery procedure */
 for( each node ∈  manyCoreSystem )
 {
     node.nodeAssign(QMS)
     for( each core ∈  node)
         core.coreAssign(discoveryEvent, RCT)
 }

 On_discoveryEvent: { query=generateQuery(applicationArguments) }
 core=getOriginCore()
 QMS=getLocalQMS(core)
 multiCastGroup=getLocalCores(QMS)
 reqMessage =generateRequestMessage(query, multiCastGroup)
 Set timing = true
 on_send:  Broadcast(reqMessage, AnyCast) //send query from orign core
 on_receive:
 { //receive query from other cores
     for ( each resource ∊ multiCastGroup )
 if  (checkQueryRequirements(reqMessage,resource)>0)
                 qualifiedResources.add(resource)


     if (qualifiedResources.length==0) {
         remoteQMS_group=getListQMS(resource.nodeID)
         for each ( remoteQMS ∊ remoteQMS_group )
         {
             multiCastGroup=getLocalCores(remoteQMS)
             reqMessage =generateRequestMessage(query, multiCastGroup)
             Broadcast(reqMessage, AnyCast)
         }
         }
     else{
         RCT=resourceRanking(qualifiedResources)
         core.setRCT(RCT)
         discoveredResource=core.getRCT.getBestMapping(query)
         DHT_resource_mapping.add(query.id, discoveredResource)
         }
 }
```

Algorithm 1: Resource Discovery Algorithm

The procedure of resource discovery takes place according to the following steps:

1. System initialization and resource description: in this step the whole many- core system has been described, the components such as QMS, RCT and Discovery Event assigned to each processing unit.
2. Once a core determined an overload execution , it triggers a discovery event to distribute extra loads over other cores in the vicinity.
3. Discovery event generates a query which requests on-demand resources from local and remote nodes . The query is generated according to the application requirements and parameters. For instance, depending on the type of an application , queries can be produced to find The most efficient resources due to Flynnś classifications .
4. Origin core is a processor that starts query dissemination. It sends a message for local QMS and checks the list of surrounding processors. The reply message includes a list of possible local resources to create a Multi-Cast group. Origin core sends Any-Cast request message to the group's address.
5. All the receiver processors check the query message and its conditions and requirements and according to their resource description if they can fulfil the conditions they will be added to a collection of the qualified resources. In the next step the qualified resources for an individual query must be evaluated and assessed by using resource ranking algorithms.
6. If local processors can not provide a specified query with the requirements and collection of qualified resources is empty, the discovery procedure checks the list of remote QMS and sends Any-Cast queries to all the remote resources neighbouring the origin core. This process will continue until the qualified resource collection is not empty.
7. After finding qualified resources for a particular query , according to a ranking algorithm these resources would be assessed and the best mapping is assigned the correspondence . each of the processors has its own RCT table to show The cost of task migration to the other processors. RCT tables are used for successive queries with the same requirements.

Figure 5 depicts the mechanism of resource discovery in a cluster with multi-core nodes. The proposed search method is acting similar to BFS, Which is due to the point of memory latency, bandwidth and interconnection network. To find the optimal resource for a query with minimal requirements which depends on processing capability, memory and availability, the search algorithm first explores the possible neighbouring nodes and ranks them according to a defined cost table from the source node to the destination, and then selects the most promising ones, explores the search graph and goes to the next tier only if the founded optimal does not meet the minimal requirements of the query.



QMS
Search in the next neighboring tires
resource(m).setrank=query(z).getorigin.getrank(m)
resourceQuery(minReq)
RCT
reply(RO)
If queue(i).lenght(i) > threshold then generate.query(minReq)

Figure 5: Mechanism of resource discovery according to the cost of resources

3. Resource Assessment Algorithms

In D2.2 [3] we have classified the performance metrics and parameters to evaluate resources in format of resource description in three individual layers: Core2Core, Die2Die and Node2Node communications level. These parameters handles gathering real time information about the current status of executions and processing capabilities of the available resources. Each processor has its own resource cost table (RCT), see Figure 6, which is includes metrics to asses other processors for movement of executions to destination processors. According to the metrics values in RCT, ranking algorithms generate rank number for every particular processors in the network. The following figure is a sample of a generic RCT.

Under the scope of S(o)OS resource description, we must identify the metrics relevant for describing the potential performance of a given resource.

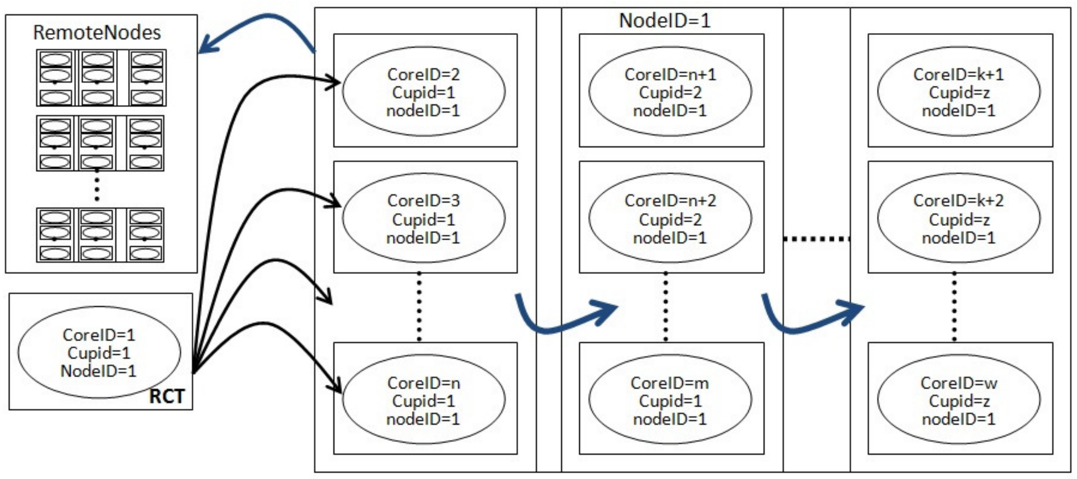


Figure 6: Resource Cost Table (RCT) assesses cost of resources per type of query for various ranges of latency

RCT aims to evaluate the cost of resources per query type. To organize RCT for execution per core we need to define query types. RCT ranks the resources for each queryTypeID which IS conceptually elaborated and IS based on the range of computation and communication latencies. Before implying query classifications we must estimate optimal or minimal application resource requirements.

A) APPLICATION RESOURCE ESTIMATION

We know that computational clusters are common alternative platforms for handling massively large computational problems and parallel applications, and the many-core systems are able to be more efficient when resource management is deployed according to the run-time application requirements . Generating queries for resource discovery requires a capability to predict the resource requirements of parallel applications before making decisions for scheduling. There are variety of researches to estimate and extract the application´s features. They are trying to deploy performance models or mechanisms to forestall resource utilization in case of computation and communication complexities for applications lunched under a distributed parallel system including multiple homogeneous or heterogeneous resources with various processing capabilities. Application modelling or application description can provide accurate information for querying operations in resource discovery protocol. Appendix B: Extraction of Application Requirements is a sample application modelling which is a non-stop loop for extracting application features for future executions.

In S(o)OS deliverable “D2.2 First Implementation Set: Hardware” [3], a section in chapter 3 identifying resource requirements from code is discussed . Here for resource discovery protocol we use a simple algorithm to categorize application features and estimate their minimal execution requirements to achieve an optimal performance.

queryTypeID	LowerBound	UpperBound	Minimal/Maximal	Processor Type
---------------	Memory Access	Memory Access	Frequency	SIMD / SISD MIMD / MISD
---------------	L1_Latency	L1_Latency	cacheSize_L1	
---------------	Run time	Run time	Pipeline stages	

Table 1: Argument for query classifications

At the time of query generation, due to the application features we can assign a queryTypeID for each particular query. And after discovering an appropriate resource, all the query-resource mapping information added to DHT_resource_mapping for task management and also reuse of the discovered resources in similar query statements.

```
On_discoveryEvent:
{ query=generateQuery(applicationArguments) }
```

b) Resource Cost Evaluations

Resource cost is subjected to three elements. The first one is the processor that is looking for other resources to augment its processing capabilities and distribute jobs among them. This processor starts the process of discovery and triggers its assigned discovery event. The second element is the application that runs on the aforementioned processor.

There are variety of applications with different specifications and requirements, to make it simple, we assume that for each particular application we can map it to a queryTypeID which the type of query describes the application execution resource requirements. The final element is the processor that would be nominated as the result of discovery in collection of qualifiedResources. The resource cost for all the members of qualifiedResources stores in RCT of the processor which it is generated the query. According to these elements and their affects on the resource evaluation we can model the complexity of assessment as:

```
Resource Cost= Communication Time + Computation Time
```

c) Resource Ranking Algorithm

We have developed a resource discovery protocol for the cluster environment includes homogenous and heterogeneous multi-CPUs and multi-cores nodes. In classic resource discovery protocols, a typical cluster environment generally defined as a Virtual Organization (VO) which consists of two types of individuals includes resources and resource directories. In our approach for discovery we assume that a cluster is a directed graph $Cg(Ve, Ed)$ with n nodes and w edges. Each of edge $ed(a,b)$ connect two nodes a and b where:

$ed(a,b) \in Ed,\ and\ \forall\ a,b\ \in Ve\ \ a,b\ \leq n$

Nodes in the graph are cores in the real cluster. Cores are linked to each other through interconnection networks, for all the cores which are placed in a CPU or cluster node an identity has been defined which Queue Management System (QMS) is. The QMS includes a set of addresses and descriptions for resources with maximum priority in resource discovery. in our graph we allocate set of first priority resources for each node.

$ed(a,b)$ is connection link between core a and core b , communication cost to describe this link are bandwidth and delay. For each core, states and descriptions of its processing resource and also other cores in the first-tier vicinity stores in localQMS. Cost of resource r in localQMS of core b through $ed(a,b)$ maintained as $\frac{1}{C_{ab}}$.

In our resource discovery approach, we generate one query per discovery request, and in the whole cluster system concurrently several query would be generated. In our ranking algorithm each of the successful queries with the best matching discovery affects the dynamic values of the resource cost and resource rank consequently. Queries explore the cluster and checks local and remote QMS for the best resource matching.

When a query explore a QMS and related resources, it shows one of the following two behaviors: first, query does not find the best matching, then it goes to the next QMS, and will keep continue until the best matching for discovery be achieved. If a query that is exploring QMS associated with node a does not satisfy with the local resources then it goes to the next node b in one of the QMS neighbor with the following probability which is proportionally related with the rate of $\mathcal{C}_{ab}$ for each one of r resources over edge $ed(a,b)$.

$$Prob^{r}_{ab}=\frac{\mathcal{C}_{ab}}{\sum_{z}\mathcal{C}_{az}}. \quad z\in(QMS\ in\ vicinity\ of\ node\ a),\ r{:}coreID,\ remote\ resource\ in\ node\ b \qquad (1)$$

According (1), queries use the probability to select the next node for discovery. After finding a match for a query, the reply message includes the description of the qualified discovered resources returned to the initiated node. Among all the discovered resources the best matching is a resource with lowest rate of latency. In the next step rank for the best matching discovered resource would be update according to following formula.

$$ResC^{r}_{ab}=\frac{1}{ResC^{r}_{ab}-\delta ResC^{r}_{ab}-\frac{\delta}{\min\left(La_{ab}\right)}} \quad ,$$

$$0<\delta \ll 1,\ \partial\ is\ random\ number,\ La_{ab}\ is\ latency\ between\ node\ a\ and\ node\ b$$

Resource rank for a collection of the qualified resources assigned according to a list of resource cost which is sorted based on ascending sort algorithm.

4. Simulation

a) Cluster Simulation

To study and analyse the performance impact of the future hardware architectures, network protocols, operating systems and parallel applications, we are required to use cluster simulators to simulate these future systems. Generally, cluster simulators are combinations of cycle-accurate discrete event multi processor full system simulators and network models or simulators; as such, in these systems frequent interactions among the potentially different parallel cycle-accurate simulators are necessary. Cluster simulators then need to support an efficient synchronization mechanism.

The main performance metrics for cluster simulators are accuracy and speed; however there is a tradeoff between these factors, it is important to verify correctness of simulation when the simulator emulates the behaviour of a real physical machine with real devices, operating systems and user applications. To increase the accuracy of the simulator we need to increase the frequency of the synchronization which will slow down the operation speed of the simulator. Synchronization is a critical challenge for current research on cluster simulators.

In some research works, they have tried to solve the synchronization problem by proposing asynchronous distributed simulation, however this is not accurate enough, and for increasing simulation accuracy costs increase: more frequent synchronization that causes limitation on performance, simulation speed and scalability. The COTSon simulator is concentrated on best combination of both, trading speed for accuracy depends on simulation purpose and the user preferences. The MARS simulator platform and BigSim are another alternative for COTSon, MARS has focused on offering a very detailed and accurate network simulator, instead a network model, it fed the network simulator by traces which are generated by IBM-Mambo, the solution scalable but has some disadvantage, as the detailed network and traffic analyzes impact on node architectures. BigSim also uses the Mambo full system simulator with an optimistic method for synchronization, for exploiting parallelism it could be efficient for fine-grained tasks but Mambo, as a single parallel task is coarse grain. BigSim is not able to simulate node detail in clusters.

Full system simulation is not a new research area, there are a lot of works about virtual machines and cycle accurate simulators have extended these works. Several system simulators such as AMD SimNow ,SimOS, Simics Central come with a built in functionality that enables them to communicate with other instance of the simulators or the real network by a network model, also many virtual machines and emulators, such as VMWare or QEMU embed similar "virtual networks" functionality, to model the networking among all the instances and other physical machines. Other than the above mentioned performance metrics, there exist other metrics for assessment of a cluster simulators such, repeatability and scalability. The regular techniques to evaluate simulated clusters are using benchmarks such as, NAMD,HPL, IS (from NAS Benchmark) , AMG (from Sequoia Acceptance Suite), LAMMP(from Sequoia Acceptance Suite), FFTW, HPCCG. In this work we have selected COTSon and SimNow simulators to evaluate the proposed resource discovery protocol.

B) *Simulation in COTSon*

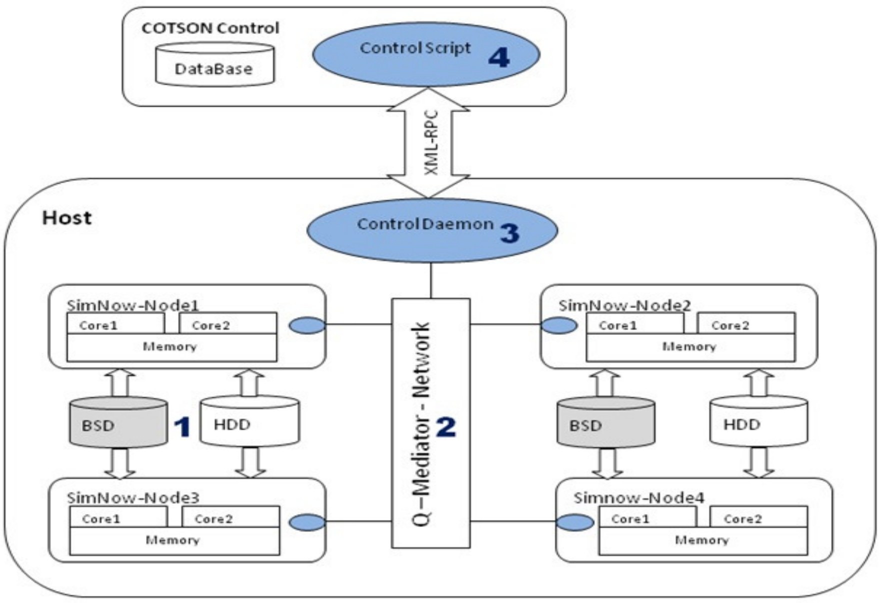


Figure 7: Simulation architecture for a cluster combined of 4 dual core simulated nodes hosted on one physical machine

We have simulated a cluster over two physical nodes by executing four instances, see Figure 7, of a simulated node (target node) on the first machine and two instances of a target node on the second machine. Each target node has two cores lunched per physical core. The simulated nodes communicate with each other through a component which is responsible to model NIC and network. This component is called mediator.

The NIC model provides facility to connect the target nodes with their related host nodes. The Q-Mediator is an application from HP that implements AMD SimNow's mediator using C++, it is a pluggable network model component that is able to be connected to SimNow service and provides functionalities for communication among multiple SimNow instances by using Ethernet frame encapsulating inside UDP packages. For the purpose of simulation we have implemented our resource discovery model by Using Open MPI (RDMPI).

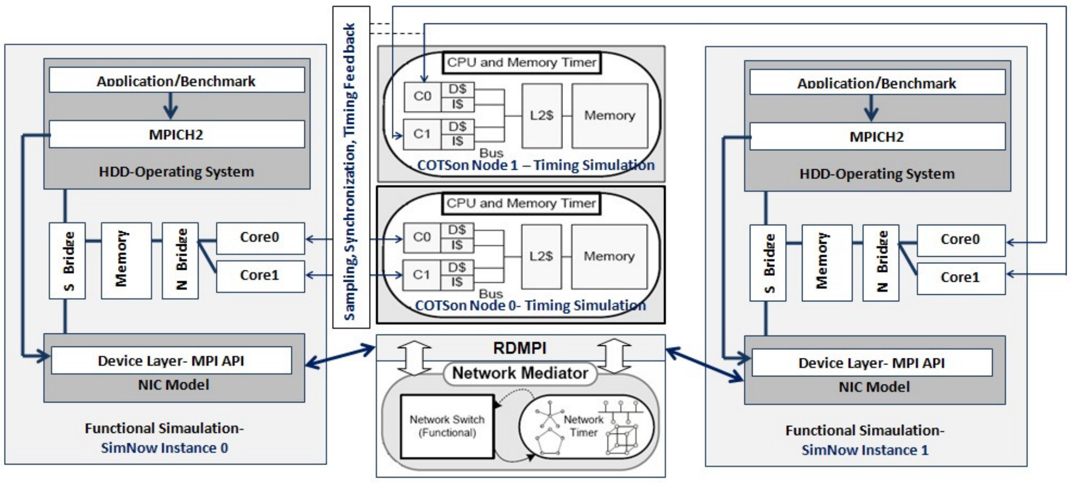


Figure 8: Interaction between simulator components COTSon , SimNow, Mediator and RDMPI

This application used Q-Mediator to manage and control communication between target nodes in simulated cluster. SimNow instances also are able to connect to the real networks by using Virtual Distributed Ethernet (VDE). RDMPI implements the resource discovery mechanisms on top of the Q-Mediator layer.

Figure 8 depicts interaction among various simulation components, we have explained our resource discovery in previous sections, RDMP is an implementation of this algorithm using Open MPI and C++. it works with Q-Mediator and took place between NIC Model in SimNow instance and Network Model. In order to simulate the proposed resource discovery method, first we established a simulator cluster with interfering of RDMPI in mediator and network transport layer. And finally we use some standard parallel application benchmark to evaluate the simulated cluster with resource discovery enhancement. In Figure 3, application uses MPICH to communicate with the simulated system. It interacts with device layer in NIC model in SimNow instance, NIC models can communicate with other NIC models by using RDMPI and mediator.

c) Functional and timing simulation

We use AMD SimNow as the simulation component for functional simulation. It is a fast cycle-accurate full system emulator and use caching and dynamic compilation technique. It can support booting an real operating system and lunch complex applications over it. The SimNow simulator deployed x86 and x86-64 instruction sets with support for system devices . It perform simulation of the real system with 10x slowdown in comparison with the native execution. SimNow cores generate a series of event that are stored in asynchronous queue. COTSon provide timing feedback for SimNow instances, it parses asynchronous queues to create higher level objects such as instruction. COTSon provides timing information back to the functional simulators to affect the behaviour of the application. It also uses Quantom based simulation for synchronization techniques, it uses a Quanta (Q) bigger than latency time between two nodes.

d) Bandwidth and latency simulation

We are simulating a cluster which is a LAN network with high bandwidth and small value of latency. Therefore simulation of cluster is easier than a real network with complex bandwidth and latency behaviours. Functional simulation generally adds some extra latency to every transmitted packet. It happens because we send packets twice , one time functional simulator sends packet to the mediator and in destination, mediator sends packets to the receiver node. COTSon performs bandwidth and network simulation in the sender NIC device, but all the network timing characteristics and information is collected in mediator level.

IV. Network Models

1. Functional Network Topologies Description

Given that future systems will have more and more components, both on-chip and off-chip, there is a need easily experiment with different, complex, topologies [4]. We need a language-based approach to these topologies, as an overview of the network is quickly lost when working in a purely graphical environment. We chose a functional languages approach to generate and manipulate these network topologies, for reasons we hope to make clear in the rest of this section.

We describe the layout/topology[13] of a network using a sparse adjacency matrix, annotated with the properties of every link. We use a sparse representation because we assume that a highly connected topology is uncommon and we envision that we will be working with topologies that have hundreds or even thousands of nodes.

```
type Layout = [[(Int,LinkProps)]]
```

The *Layout* type is defined as a list of lists, where the inner lists represent the rows in the sparse adjacency matrix. Every element in a row is tuple consisting of the index of the *source-node* and the link properties of that incoming connections. Because we annotate each connection with link properties, it becomes possible to have different up- and downstream properties. The link properties currently captured in our topology model are bandwidth and latency. Extending them with information regarding reliability and availability is considered as future work.

```
 type LinkProps = (Bandwidth,Delays)
```

Although the concept of using a sparse matrix to describe a layout can be captured in almost any modelling approach / programming language, we chose to use a functional language to manipulate and generate topologies. Support for built-in dynamic list structures, polymorphism, higher order functions, and lazy evaluation, leads to very natural descriptions and generators of complex topologies. We mean by 'natural' description, descriptions "in code" that have an almost one-to-one mapping to the descriptions "in words". We will illustrate this concept using a fairly simple topology: a fully-connected layout. In a fully-connected layout each node is connected to every other node. The function "fullyConnected" has two arguments: the number of nodes in the topology, and a link property that will be associated with every link.

```
fullyConnected :: Int -> LinkProps -> Layout
fullyConnected n lp =
  [ zip (filter (/=k) [0..(n-1)]) (repeat lp) | k <- [0..
(n-1)] ]
```

where the *zip* function zips two lists together to form one list of tuples, the *filter* function removes every element from a list that does not pass the given test, and the *repeat* function creates an infinite list of the given element. The *fullyConnected* function uses a so-called *list comprehension*, of the form *[f e | e <- g]*, to create a list of *e*'s generated from generator *g*, where a function *f* is applied to every *e.* In the case of *fullyConnected* this generator is the list *[0..(n-1)]*, which is syntactic sugar for the list of integers from 0 to (n-1).

The above list comprehension will create *n* rows, and each row, *k*, will consists of the neighbours of *k*. The neighbours of *k* are all nodes 0 to (n-1) except *k* itself (hence the use of the *filter* function).

13 The use of the terms 'topology' and 'layout' in this section both refer to the concept of 'topology'.

We also need to associate the link property *lp* with every link to a neighbour, for which we use the *zip* function. We noted earlier that *repeat lp* will create a list of infinitely many copies of *lp*, this will however not cause any problems due to lazy evaluation. The *zip* function will only evaluate as many copies created by *repeat lp* as there are elements created by *filter (/=k) [0..(n-1)]*.

We feel the description "in code" follows natural from the description of the fully-connected layout "in words": every node is connected to every other node, and all links have the same property. The list comprehension captures the "every node", the *filter (/=) [0..(n-1)]* captures connected to every other node", and the combination of *zip*, *repeat*, and lazy evaluation corresponds to "all links have the same property". Where the use of lazy evaluation is particularly convenient, in that we only create as many link properties as needed.

A) Combining topologies

A more interesting application follows naturally from the functional approach is combining several existing topologies to form a new topology. We highlight one layout combinator, the *ring* combinator (see "Appendix A: Ring layout combinator" for the actual code), that combines several given identical layouts (with n nodes) by putting a ring through the corresponding nodes, for every node in these given layouts. The (bidirectional) rings that connect the layouts all have the same link properties. The type signature of *ring* is thus:

```
ring :: LinkProps -> [Layout] -> Layout
```

To create a simple ring network of *n* nodes, we can call the *ring* layout combinator on a set of *n* layouts that have a single node with no connections. We define such a singleton layout as follows:

```
singleton :: Layout
singleton = [[]]
```

Going back to the definition of *Layout*, we see that the adjacency matrix of *singleton* has one row (a single node) that is completely empty (*[]*), indicating it has no connections. Having the *singleton* layout, defining the ring topology becomes straightforward:

```
ringLayout :: LinkProps -> Int -> Layout
ringLayout lp n = ring lp (replicate n singleton)
```

where *replicate n singleton* makes a list of *n* copies of the *singleton* layout. Figure 9 shows the result of generating a layout using *ringLayout (2,4) 5*, to create a ring topology of 5 nodes where the up and downstream link properties are identical.

Now that we have a function to generate ring topologies, the natural extension is to combine several ring topologies to form a 2D torus. We will use the *ring* layout combinator again to create an *n* x *m* 2D torus, by combining *n* rings that all have *m* nodes.

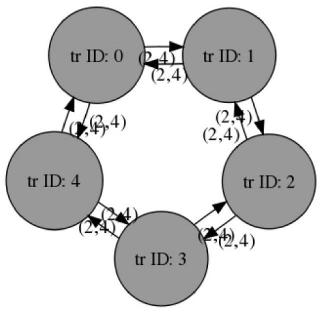


Figure 9: Ring layout of 5 nodes

```
TwoDTorusLayout :: (LinkProps,Int) -> (LinkProps,Int) -> Layout
twoDTorusLayout (lp1,n) (lp2,m) =
  ring lp1 (replicate n (ringLayout lp2 m))
```

When we generate a layout using *twoDTorusLayout ((6,8),3) ((2,4),5)* to create a 2D torus consisting of 3 rings of 5 nodes each we get the topology we see in Figure 10.

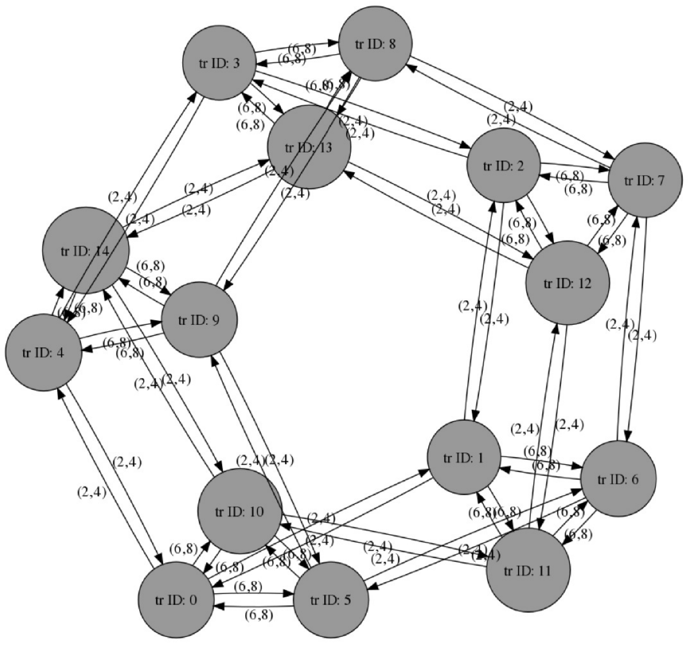


Figure 10: 3x5 2D Torus Topology

We were able to create a 2D torus by combining several ring topologies using the *ring* layout combinator. We can use the *ring* layout combinator again to create a 3D torus topology by combining several 2D torus topologies. Actually, we can make a N-dimensional torus by calling the *ring* combinator recursively on a set of (N-1) dimensional torus topologies. The zero-dimensional torus is then the earlier mentioned *singleton* layout.

```
nDimenTorusLayout :: [(LinkProps,Int)] -> Layout
nDimenTorusLayout []           = singleton
nDimenTorusLayout ((lp,n):dim) =
  ring lp (replicate n (nDimenTorusLayout dim))
```

where the code in the form *(x:xs)* refers to a list where *x* is the first element, and *xs* is the rest of the list. When calling the above *nDimenTorusLayout* function with the list *[((2,3),5),((2,3),6),((2,3),4), ((2,3),8),((2,3),9)]* we get a 5-dimensional 5x6x4x8x9 torus. We do not show a picture of this 5-dimensional torus as the flattened, 2D representation, is both too large for this document, and offers no real insight of the generated structure.

As both the ring topology and the 2D torus topology are both two specific instances of the more general N-dimensional torus topology, we can also redefine the *ringLayout* and *twoDTorusLayout* function in terms of the *nDimenTorusLayout* function:

```
ringLayout' lp n = nDimenTorusLayout [(lp,n)]

twoDTorusLayout (lp1,n) (lp2,m) =
  nDimenTorusLayout [(lp1,n),(lp2,m)]
```

Encouraged by the versatility of the *ring* layout combinator, allowing us to generate complex structures in a clear and unencumbered way, we intend to investigate other layout combinators in future iterations. We also intend to make a translator that converts the network topologies created and described in this section, to topology descriptions that are used by NoC-simulator “McoreSim” described in project deliverable D2.2 [3].

2. The MCoreSim NoC Simulator

As mentioned in the S(o)OS Project Deliverable D2.2 [3], we are developing MCoreSim [5], a Many-Core Simulation framework that aims to allow for comparing different possible alternatives in future many-core, distributed and massively parallel systems. The simulator was born with the main objective of simulating the impact of the NoC latencies for the purpose of evaluating different strategies for common operations that are recurrently performed by the software stack, either through the Operating System, or by application-level middleware, such as basic low-level synchronization and communication primitives for Inter-Process Communications (IPC), as well as scheduling and deployment policies. MCoreSim is based on the well-known OMNeT++ framework for network modeling and simulation. In this section, we focus on the MCoreSim Protocol Library, which implements in the MCoreSim Communication Plane the simulation of the in-chip communication capabilities foreseen in future many-core systems.

a) MCoreSim Protocol Library

The MCoreSim protocol library provides the protocols for simulating communications and computations in MCoreSim. The protocols planned to be implemented in the MCoreSim Communication Plane are the network-on-chip protocols related to buffer management, switching, arbitration, packet scheduling and routing.

Up to now the library implements protocols related to a basic communication and computational model. MCoreSim supports mesh and torus regular topologies, and the popular deterministic routing Dimension-Order Routing (DOR) [6]. In this routing, no storage element like register file or routing table is required.

There are various switching protocols available but, as the number of cores increases, the on-chip transmission bandwidth available to each core decreases. Due to this constraint, we implemented in MCoreSim wormhole switching [6]. In this approach, a Network Interface splits a processor Instruction packet into smaller size data units called Flit, then it transfers them towards destination through the network. At the destination tile, the Network Interface re-assembles the processor Instruction packet and sends it to the PE for further execution.

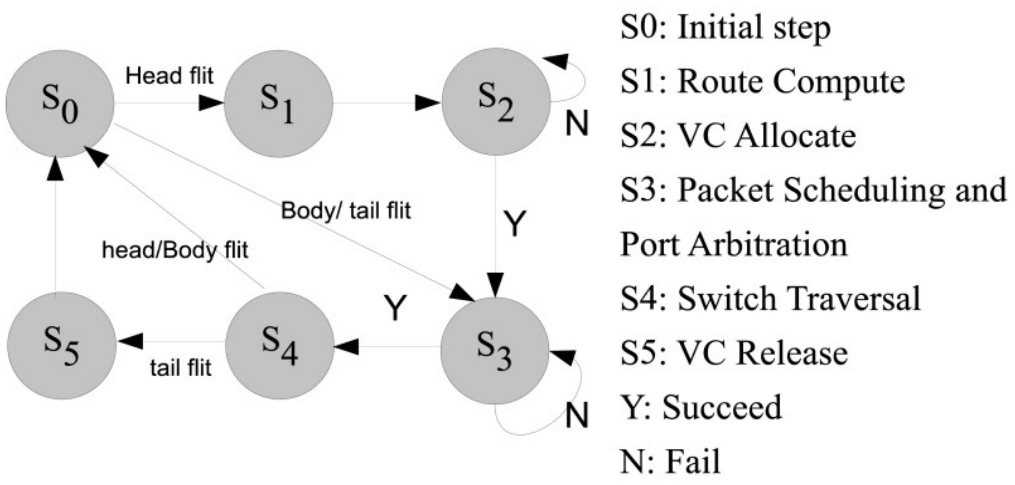


Figure 11: Sequence Chart of Router.

Between the source and destination NIs, routers handle Flit packets in the following manner (see in Figure 11). The Route Computation (RC) module computes route according to the specified routing algorithm, the Virtual channel allocation (VC) module allocates a virtual channel for the next router input buffer, as specified by the RC module, and according to the Virtual channel allocation algorithm. Then, the Switch Arbitration (SA) module, based on packet scheduling and port arbitration algorithms, sets the path up from the input buffer to the output port in the Switch Traversal (ST) module.

There are various flow control protocols available for switch-to-switch or end-to-end transmissions. In MCoreSim, a credit-based Link-level Flow Control protocol is implemented. This credit flow control protocol is employed in between network interface (NI) and router, and among routers. The credit-based flow control mechanism indicates to the output port the buffer space availability in the adjacent input ports. Once an output port runs out of credits, it will stop sending flits to the adjacent input port.

The MCoreSim Router element has a buffer for each input channel port. For buffer management, the Router implements a virtual channel based protocol. In this each physical channel is divided into a number of virtual channels and each virtual channel has its own buffer space. After the route computation, a request is made to the virtual channel allocator (VC) to allocate an unused virtual channel to the new route. After the allocation, the switch arbiter (SA) schedules the packets that flow through the output port of the Router.

Among the variety of arbitration algorithms available, in MCoreSim only FIFO based port arbitration is implemented as of now.

V. State Maintenance

1. Message Passing

In this section, we focus on message passing in shared memory, which has recently attracted significant attention as a way to build scalable applications that enable applications to execute transparently on heterogeneous systems that are becoming more widespread. In particular, we describe how to implement message passing at the user level on cache coherent and non-cache coherent systems.

A) Non-Cache Coherent Systems

The increase in the number of cores tends to shift the way we envision communication in operating systems towards a more inherently distributed systems.

We are currently investigating the performance of distributed atomic operations executed as *transactions*. Deliverable 4.2 presents our recent achievements in this context by providing a new Distributed Transactional Memory (DTM) algorithm dedicated to non-cache-coherent architectures. Below, we present the underlying protocols used for communication (by this DTM) as an alternative to the cache coherent protocols that may not scale up to many cores.

Intel Single-Chip Cloud Computer (MPB)

The Single-Chip Cloud Computer (SCC) experimental processor is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research. The 48 cores are arranged in a 6x4 on-die mesh of tiles with two cores per tile. SCC provides no hardware cache coherency and the nodes communicate via message passing. Drawing 4 shows the available memory address spaced on the SCC.

Message Passing Buffer (MPB)

In addition to the traditional cache structures, a local memory buffer (MPB) capable of fast Read/Write operations has been added to each tile. This 16KB buffer provides the equivalent of 512 full cache lines of memory. Any core or the system interface can write or read data from these 24 on-die message buffers. The intended use for the MPB is fast core-to-core message passing. The data that are sent via the MPB have a special data type, called MPDT. All accesses to MPDT data bypass the L2 cache.

The RCCE Message-Passing Library

RCCE runs on the SCC chip, as well as on top of a functional emulator that runs on a Linux or Windows platform that supports OpenMP. Communication in RCCE is synchronous (blocking) and deterministic (for every send request a corresponding receive has to exist and vice-versa).

The iRCCE Library

iRCCE is a extension of RCCE Library which provides asynchronous, but still deterministic, message-passing functions. iRCCE also allows multiple outstanding requests and provides functions for testing their completion. Non-Deterministic Asynchronous Communication In our DSTM algorithm (presented in *Deliverable 4.2*) the communication is not predefined, meaning that the send and receive requests cannot be simply coupled since at any time a message may arrive from any other core. In order to bypass this limitation, the pending request queues of iRCCE were used to implement the communication as following. Every node performs the following steps in a loop:

Keeps a pending receive request for every other node in a waitlist and checks the waitlist for incoming messages. On asynchronous send, adds the send request (if not

completed immediately) in a sendlist. Checks the sendlist for completed send requests.

B) CACHE COHERENT SYSTEMS

The main purpose of using message passing on cache coherent systems is to enable scalability and operation on heterogeneous hardware.

1. **Scalability.** Using message passing enables writing of scalable software as it makes all communication between various threads and processes of the applications explicit. With the more traditional approach to concurrent programming on the cache coherent machines, which includes well-known locking and lock-free techniques, the communication between the threads and the processes of the application is explicit as it is done as a part of the cache coherence protocol. With this approach, threads share some data and when they access the data, the up-to-date versions of the data are passed between CPUs transparently to the application. This can often lead to scalability and performance issues as the communication between the CPUs becomes overwhelming. On the other hand, when the communication is done explicitly to the messages, it is limited to only the data that needs to be passed between the threads and processes. This has the potential of reducing the cache coherence traffic at higher levels of parallelism and improving the scalability of the applications.
2. **Heterogeneous hardware.** When the communication between threads and processes includes message passing it becomes hidden in the communication layer and transparent to the application. This enables some parts of the application to execute on a local CPU, others on another CPU in the same machine, yet others on the GPU in the same machine and some even on geographically distant machines completely transparently to the application code. In this section, we focus on the communication only between CPUs on the same physical machine and leave the more interoperability-centric message passing system for future work.

SWISSMP MESSAGE PASSING

We developed a user-level message passing library called SwissMP. The main goal of this library is to enable light-weight communication between the application threads that reside on the same physical machine.

Communication model

In order to provide light-weight message passing, the library provides a polling interface for threads that receive incoming messages. The usage of polling as a message passing interface enables us to avoid overheads of context switching and achieve high communication performance.

Alternatively, the library can be used with the cooperative user level threads to create an illusion of asynchronous message passing if such a model is more appropriate for the application. We still have not integrated SwissMP with any of the user-level threading libraries, but we do not expect any problems with such an integration. The current polling approach was completely satisfactory for the current applications of SwissMP.

In SwissMP, messages are passed between communication *endpoints.* Each endpoint has a unique *address*, which is an integer value assigned by SwissMP library. Each thread can own one or more endpoints, although we expect that in typical uses of SwissMP each thread will own only one endpoint. SwissMP also provides an API call for assigning endpoint to the OS CPU. This information is used internally in the SwissMP to enable construction of efficient multicast and broadcast trees, but it does not actually tie a thread to a particular CPU – the user has to invoke the appropriate system call to do so. We decouple these interfaces to avoid unexpected interactions of SwissMP with other parts of the system and application.

In a typical execution of applications that use SwissMP, each thread creates (or is assigned) one endpoint during the initialization and uses it subsequently for the communication. Currently, the mapping between threads and endpoints is done outside the SwissMP library. This can be done by using a separate mapping shared data structure that is effectively read-only after the application initialization phase. After initialization is done, each thread can use the assigned endpoint to send the messages to other endpoints (effectively other threads in the described setup) and to receive messages from them. SwissMP needs to be extended further to support dynamic subscription of new endpoints to the system. We do not envision that such an extension would change anything in the SwissMP dramatically, but we still did not implement it.

Messaging queues

During the initialization, a pair of queues is created between each pair of endpoints (Figure 12). Each of the queues is used for one-way communication between the endpoints. With such an organization, for each of the queues the elements are enqueued by one thread and dequeued by another one. This enables us to use simple sequential cyclic queues, where only one thread updates the head index (but reads the tail) and when another thread writes the tail index (but reads the head). Such a queue can be implemented without using any locking or any of the expensive atomic CPU instructions (such as compare and swap). We only need to use a memory ordering barrier to ensure that the published data has actually be written to the queue, before the head index has been updated and that the consumed data has actually been read from the queue before the tail index has been updated.

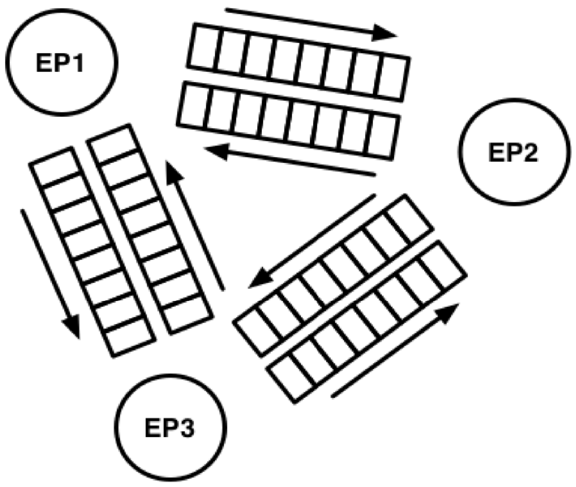


Figure 12: SwissMP message queues between three endpoints

To improve cache efficiency of the queue we keep the head and the tail indexes in separate cache lines. Furthermore, we also keep the actual queue data aligned to cache line boundaries (Figure 13). Each message can span multiple cache lines, but it is always padded up to the next cache line boundary to avoid false sharing and multiple unnecessary cache line invalidations while the next message is being enqueued to the queue. The first two bytes of each message specify the message size in bytes, as the message sizes can vary.

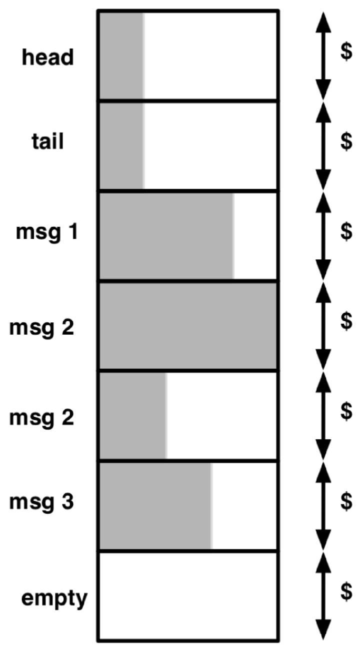


Figure 13: Cache line organization of the SwissMP queues. Each cache line is represented as a rectangle. The shaded area of the cache lines represents the occupancy of each cache line. There are three messages in the queue with one empty cache line.

With this message queue organization, the number of cache invalidations for sending or receiving a message that is m cache lines long is m + 1. There is one cache line invalidation for writing or reading each of the message cache lines and an additional one for updating the head or the tail index of the queue.

The downside of our design is that the time required to receive the message increases linearly with the number of the endpoints registered in the system. Alternatively, we could use a single receive queue per endpoint, which would keep the receive time constant, but would increase the complexity and required time for sending a message as a more elaborate and time consuming synchronization scheme would be needed. Also, the number of the cache invalidations per data cache line sent would vary and would be higher than with the current approach.

Unicast

Sending a unicast messages consists of the following steps:

1. Locate the queue used for sending the message to the receiver. This step is rather simple as it only requires a simple array indexing.
2. Ensure that there is enough empty space in the queue. This requires reading both the head and the tail of the queue.
3. Copy the message length into the first two bytes of the message in the queue.
4. Copy the rest of the message to the queue.
5. Update the head index to point to the start of the next message to be sent. A release memory barrier is required at this point to prevent the recipient from reading the data written in an incomplete step 4.

Receiving a message from any of the endpoints consists of the following steps:

1. Loop through all queues that correspond to endpoints registered in the system and are used to send the messages to this endpoint.
2. If all queues are empty return the indicator that there are no messages sent.
3. Otherwise, read the size of the message from its first two bytes.
4. Copy the message data to the buffer passed from the application. The buffer should be large enough to receive all the application messages. Allocating the large enough buffer is the responsibility of the caller.
5. Update the tail index to point to the start of the next message to receive. A release memory barrier is required at this point to prevent the sender to overwrite the contents of the message that is being read in an incomplete step 4.

Multicast and broadcast

SwissMP supports efficient message broadcast and multicast by exploiting the CPU and cache organization of the machine. In a nutshell, broadcast and multicast trees are built according to the CPU and cache organization and messages are sent and forwarded along the trees by the participating endpoints. An example machine configuration and the corresponding broadcast tree are depicted in Figures 14 and 15. For this, the message header is extended to include the originating endpoint and the forwarding information. Using the broadcast tree for a machine consisting of 48 cores, organized into four chips of two CPUs each where the CPUs consist of six cores that share a L3 cache (the machine is two times larger than the machine depicted in Figure 14), we were able to improve the broadcast performance by around 10x over a simple sequential broadcast and around 10% over the tree-based broadcast that uses non-cache hierarchy aware trees.

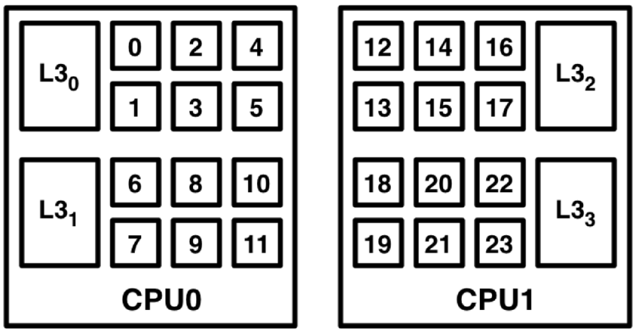


Figure 14: Example machine configuration with two CPUs each with 12 cores. The cores are organized in groups of six that share the same L3 level cache.

The multicast send and receive functions are similar to the unicast ones except that they need to be extended to send and forward the data along the correct parts of the trees. When tree-based broadcast is used, a nice property of ordered delivery of messages sent from the same source is lost. If this property is still needed, the messages need to be reordered either by a communication layer higher than SwissMP using the message sequence numbers or the messages following the broadcast should be sent only after the acknowledgement of the received message is received.

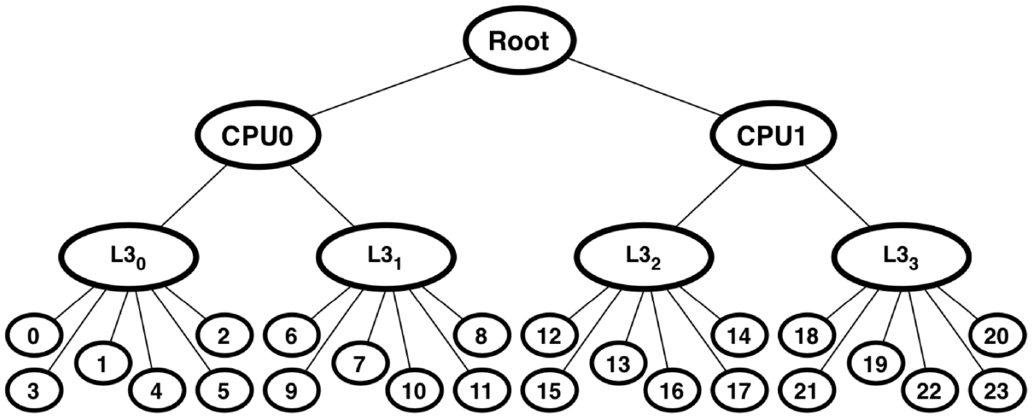


Figure 15: Broadcast tree for the example machine. The messages are exchanged between the cores that share L3 cache directly and the tree is used to reach the other CPUs.

Delivered code

We delivered source code for the SwissMP library and several simple micro-benchmarks as a part of this work package. The micro-benchmarks are used to demonstrate how SwissMP can be used in several simple scenarios as well as to measure the performance of the basic messaging primitives. The following micro-benchmarks are built:

1. **p2p** is a benchmark that is used for measuring the latency of sending messages from one thread to the other at the highest possible speed. The exact number of messages and the CPUs to which the threads and endpoints are assigned can be specified as command line arguments.
2. **p2p_2way** is a benchmark that is used for measuring the latency of two-way communication. Two threads are used and the first endpoint initiates the communication by sending a message to the second one and then waits for the response from the second thread. The exact number of messages and the CPUs to which the threads and endpoints are assigned can be specified as command line arguments.
3. **Broadcast** s a benchmark that is used for measuring the latency of sending broadcast messages from one thread to all the other participating threads at the highest possible speed. The exact number of participating threads, messages and the CPUs to which the threads and endpoints are assigned can be specified as command line arguments.
4. **mp_lock** is a simple implementation of an exclusive lock using SwissMP. One thread serves as a server that implements a lock and executes the request from all the other participating threads. The exact number of participating threads, lock/unlock operations and the CPUs to which the threads and endpoints are assigned can be specified as command line arguments.
5. **cas_lock** is an implementation of compare and swap based lock. It is used to compare the performance of traditional lock implementations to the message passing one.
6. **mcs_lock** is an implementation of MCS lock. It is used to compare the performance of traditional lock implementations to the message passing one.

Both SwissMP and micro-benchmark bundles contain a simple README file that describes how to build and run them. Both components use open-source atomic_ops library. Standard make is used for building them.

2. Replication

The communication paradigm at the heart of operating system kernels will have to be redesigned to fit the scalability requirements imposed by upcoming concurrent machine. Although nowadays operating systems are embracing concurrency on multiple cores using traditional cache coherent techniques, it is likely that future systems will suffer high contention from 10 to 20 years down the road, when many-core clusters will become the norm.

There are two distinct communication paradigms that operating systems can exploit: message passing and shared memory. A recent OS proposal, called Barrelfish, which aims at exploiting multi-core environments, has shown the benefit of message passing communication paradigm over shared memory. The shared memory paradigm penalizes seriously the efficiency of modifications in largely concurrent environments. More precisely, the authors of Barrelfish, have shown that probability of cache misses due to cache coherency is so high when 16 cores communicate through shared memory, that it can slow down by a factor of 400 a data update when compared to a sequentially running update. In contrast, considering multiple cores as distributed entities that communicate through message passing—typically by remote procedure calls (RPC)—can speedup share memory updates by a factor 2.

Besides the message passing communication paradigm, higher level communication protocols also play a significant role in congestion, depending on the number of replicas they target. For instance, the Barrelfish OS uses a two-phase commit (2PC) synchronization protocol that relies on multicast, technique mentioned in Section II, to guarantee synchronization among distributed replicas.

More precisely in Barrlefish, critical data is stored at each core. This data needs to be kept consistent among the replicas, so the cores communicate through messages in order to execute a 2PC agreement protocol. There are several drawbacks to this choice of maintaining replicated data consistent. There might be a huge penalty if, for example, there is a slow core. Also, the state is replicated at each core, which is much less efficient compared to the case when there are only a small number of dedicated server processes storing the replicated state. Barrelfish achieves efficient message passing by using multicast trees, which is constructed based on the latency of the communication between pairs of cores. However, as loads on the cores change, a multicast tree may no longer be as efficient as it was initially.

SYNCHRONIZATION PROTOCOLS

A better alternative that can ensure availability and scalability of future distributed systems while guaranteeing consistency is to run an agreement protocol. One of the best-known and most widely used non-blocking consensus protocols is Paxos. In Paxos, nodes can be of three types: proposers, acceptors and learners. Proposers try to impose the requests sent by the clients, acceptors decide which proposal to pick if there are multiple choices and learners execute the agreed requests and update the state if necessary.

Paxos assumes there is a leader process among the participants.

- In the first phase of the algorithm, the processes decide on a leader.
- The second phase then consists of the leader sending the message to be decided to the acceptors,
- and the acceptors sending it forward to the learners.

Before the learners execute a request, they need to receive messages regarding that request from a majority of acceptors. Using Paxos, messages can be totally ordered among a number of nodes, even if some participants fail.

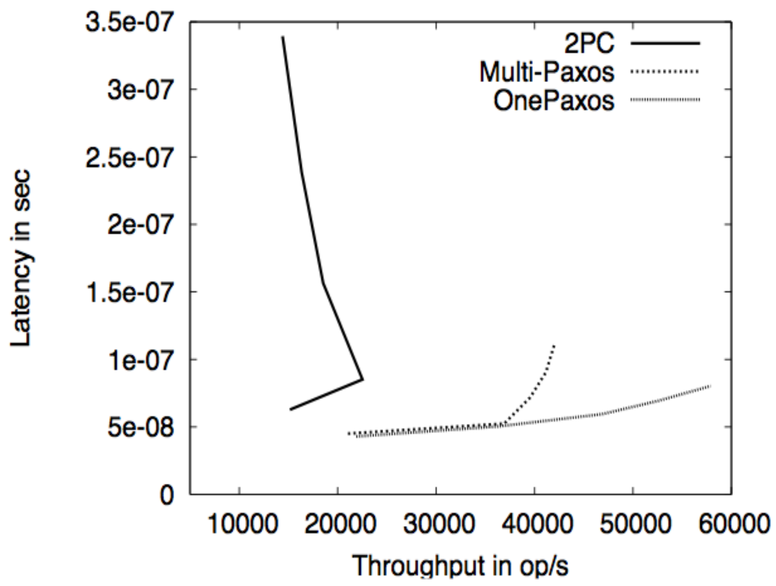


Figure 16: Paxos is more scalable than 2PC for synchronization.

We ran some experiments on a dual-quad core AMD Opteron using 3 replicas and between 1 and 5 clients to have an overview of the performance one could expect from using Paxos instead of 2PC for synchronizing replicas.

Figure 16 depicts the latency of our new protocol (OnePaxos), a variant of it (Multi-Paxos) and the usual 2PC as the throughput increases. It outlines that our algorithm scales much better than 2PC for synchronizing replicas. First, the latency remains almost constant when changing the number of clients while the latency of 2PC explodes. Second, the throughput increases with the increasing number of clients.

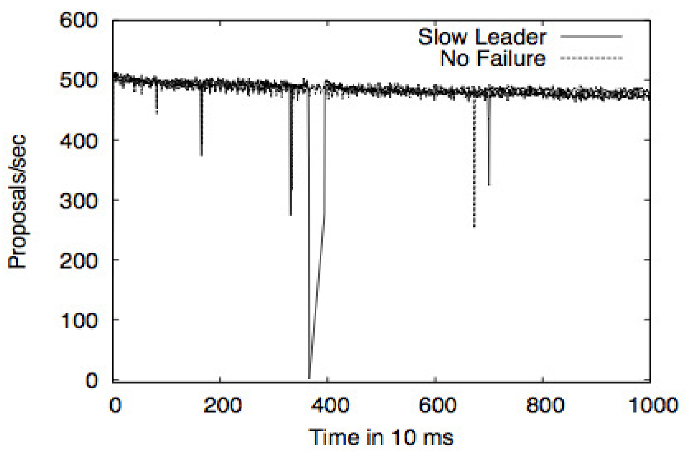


Figure 17: The Paxos protocol tolerates slow processors

Figure 17 depicts the throughput of Paxos while Figure 18 depicts the throughput of 2PC. In both cases, a slow processor is introduced. As Paxos has the ability to elect a new leader to run other instances of the protocol, it recovers from the slow-down by detecting it and choosing automatically a new leader. As 2PC relies on the participation of all nodes, performance drops as soon as a single processor slows down. This experiment indicates how availability of the synchronization service remains ensured when using Paxos instead of 2PC.

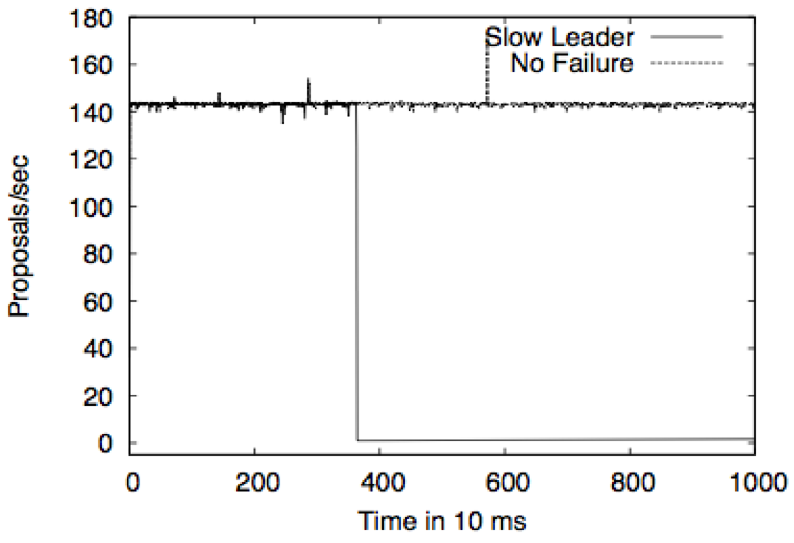


Figure 18: The 2PC protocol does not tolerate slow processors.

References

[1] *Schmidt, Douglas C.: "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", Pattern Languages of Program Design, Addison-Wesley, 529–545, Eds: Coplien, James O., and Schmidt, Douglas C., 1995*

[2] *Pyarali, Irfan, Harrison, Tim, Schmidt, Douglas C., and Jordan, Thomas D.: "Proactor - An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events", The 4th Pattern Languages of Programming Conference, September 1997*

[3] *Christiaan Baaij, Jan Kuper, Javad Zarrin, Lutz Schubert, Tommaso Cucinotta, Daniel Rubio Bonilla.: First Implementation Set: "Hardware" - Project Deliverable D2.2*

[4] *Tommaso Cucinotta, Giuseppe Lipari, Rui Aguiar, João Paulo Barraca, Bruno Santos, Jan Kuper, Christiaan Baaij, Lutz Schubert, Hans-Martin Kreuz: Definition of Future Requirements – Project Deliverable D5.2*

[5] *Sunil Kumar, Tommaso Cucinotta, Giuseppe Lipari.: A Latency Simulator for Many-core Systems, in Proceedings of the 44th Annual Simulation Symposium (ANSS 2011), part of the Spring Simulation Multiconference (SpringSim'11). Boston, April 2011*

[6] *Tommaso Cucinotta, Giuseppe Lipari, Dario Faggioli, Fabio Checconi and Sunil Kumar, Rui Aguiar, João Paulo Barraca, Bruno Santos and Javad Zarrin, Jan Kuper and Christiaan Baaij, Lutz Schubert and Hans-Martin Kreuz, Vincent Gramoli.: State of the Art – Project Deliverable D5.1, July 2010*

APPENDIX A: RING LAYOUT COMBINATOR

```
ring :: LinkProps -> [Layout] -> Layout
ring lt layouts =
  zipWith (\ls (u,d) -> (u,lt):(d,lt):ls) (concat layouts') newConnections
  where
    newConnections          = zip upConnections downConnection

    upConnections           = rotate (negate $ length $ head layouts)
                               [0..(totalLength-1)]
    downConnection          = rotate (length $ head layouts)
                               [0..(totalLength-1)]

    (totalLength, layouts') = mapAccumL renewUnique 0 layouts

    renewUnique n layout    = ( n + length layout
                              , map (map (\(k,l) -> (k+n,l))) layout
                              )
```

APPENDIX B: EXTRACTION OF APPLICATION REQUIREMENTS

Sample pseudo-code for extraction of the application requirements under cores computation and communication pattern

```
MPI Init(&argc, &argv);
MPI Comm size(&size);
MPI Comm rank(&rank);
n=size; req=Appfunc(arguments);
while(1)
{
  size_of_block=4;
  half_block=size_of_block/2
  for(k=1; k<=log(n);k++)
  {
    chk=rank mod size_of_block
    bnd=(chk <= half_block) and (chk> 0)
    if (bnd)
    {
      buffer=req;
      next=rank+half_block;
      MPI Send(buffer,next);
      req=AppFunc(arguments);
    }
    else
    {
      origin=rank-half_block;
      MPI Recv(buffer,origin);
      req_list=buffer;
      Apply(req,reqlist);
    }
    size_of_block=*2;
  }
}
```