Technische Universität München

Chair For Computer Technology and Computer Organisation

Guided Research

A Protocol for the Integration of Invasive Resource Management into Standard Batch Schedulers

Supervisor:
Prof. Dr. Michael Gerndt

Student:
Nishanth Nagendra

Advisor:
M.Sc. Isaias Alberto Compres Urena

October 7, 2015

Acknowledgement

I would like to thank Isaias for his constant support and guidance throughout the guided research especially when I faced some challenges in understanding certain topics. I would also like to thank Prof. Dr. Michael Gerndt for providing me with this opportunity to work on such an interesting topic.

Abstract

Invasive computing is a novel paradigm for the design and resource-aware programming of future parallel computing systems. It enables the programmer to write resource aware programs and the goal is to optimize the program for the available resources. Traditionally, parallel applications implemented using MPI are executed with a fixed number of MPI processes while submitting to a HPC(High Performance Computing) center. This results in a fixed allocation of resources for the job. Newer techniques in scientific computing such as AMR(Adaptive mesh refinement) result in applications exhibiting complex behavior where their resource requirements change during execution. Invasive MPI which is a part of an ongoing research effort to provide MPI extensions for the development of Invasive MPI applications will result in evolving jobs at runtime supporting such AMR techniques in HPC centers. Unfortunately, using only static allocations result in the evolving applications being forced to execute using their maximum resource requirements that may lead to an inefficient resource utilisation. In order to support such parallel evolving applications at HPC centers, there is an urgent need to investigate and implement extensions to existing resource management systems or develop an entirely new one. These supporting infrastructures must be able to not only handle the evolving jobs but also the legacy rigid/static jobs intelligently and hence newer protocols for integration of such invasive resource management into existing standard batch systems needs to be now explored.

Contents

List of Figures

1 Introduction

Invasive computing is a novel paradigm for the design and resource-aware programming of future parallel computing systems. It enables the programmer to write efficient resource aware programs. This approach can be used to allocate, execute on and free resources during execution of the program. HPC infrastructure like Clusters, Supercomputers execute a vast variety of jobs, majority of which are parallel applications. These centers use intelligent resource management systems that should not only perform tasks of job management, resource management and scheduling but also satisfy important metrics like higher system utilization, job throughput and responsiveness. Traditionally, MPI applications are executed with a fixed number of MPI processes but with Invasive MPI they can evolve dynamically at runtime in the number of their MPI processes. This in turn supports advanced techniques like AMR where the working set size of applications change at runtime. These advancements entail an immediate need for stronger and intelligent resource management systems that can provide efficient resource utilization at HPC centers. They should also now be able to achieve much higher system utilisation, energy efficiency etc. compared to their predecessors due to elasticity of the applications.

Under the collaborative research project funded by the German research foundation(DFG) in the Transregional Collaborative Research Centre 89(TRR89), research efforts are being made to investigate Invasive computing approach at different levels of abstraction vertically right from the hardware up to the programming model. Invasive MPI is one such effort towards invasive programming with MPI where the application programmer has MPI extensions available using which he/she can specify at certain safe points in the program, triggers that allow for elasticity which means the application can evolve.

1.1 Resource Management

In order to support such parallel evolving applications at HPC centers there is an urgent need to investigate and implement extensions to existing resource management systems or develop an entirely new one. These supporting infrastructures must be able to handle the new kind of evolving jobs/applications and the legacy rigid jobs intelligently keeping in mind that they should now be able to achieve much higher system utilisation, energy efficiency etc compared to their predecessors due to the elasticity of the applications. Two of the most widely used resource managers on HPC systems are SLURM and TORQUE. A resource manager together with a batch scheduler forms what is called as a Batch System that is complex piece of middleware used for managing supercomputing resources.

1.2 Batch Scheduling

The batch scheduler accepts job descriptions given by end users some of which mention as to how long the job would run and the amount of resources it will need. It maintains a queue of jobs and dispatches them to the process manager based on some criteria and algorithms. The decisions made depend on the state of resources and also others like job priorities, fairness, waiting times etc. The process manager on the other hand has lesser intelligence and does the task of mapping the processes of a parallel application on the hardware based on the node list provided to it by the batch scheduler. In the context of invasive computing we need to be investigate for new requirements in the interaction between the batch scheduler and process manager. The decisions made by the batch scheduler need to be influenced to support evolving jobs.

In contrast to the earlier uni-directional communication from batch scheduler to process manager, we now need to support a bi-directional communication between the two. The capabilities of existing batch schedulers could be leveraged rather than having to replace an entire system with a new one. The possibility of supporting a new interface for the existing batch scheduler needs to be explored such that it communicates with a new invasive process management that controls a dedicated partition to support invasive computing. An investigation needs to be done on whether the existing interface of batch schedulers towards process manager could be extended or re-used and also on the possiblity of receiving feedback from the invasive process manager to allow the batch scheduler to be influenced. The invasive process manager one level below in the hierarchy will work on local metrics of the dedicated invasive partition within the cluster. In addition to this it will also perform some kind of a run time scheduling for pinning the jobs to the nodes in the partition and also run time tuning of the parallel applications. Due to this it is more than just being a process manager and will therefore be referred to as Invasive Resource Manager.

The investigations of this guided research are an initial study that will support the continuing research effort for developing Invasive MPI and extended resource management systems to support invasive computing.

2 Invasive Computing

The throughput of supercomputers depend not only on efficient job scheduling but also on the type of jobs that form the workload. Malleable jobs are most favourable for a cluster as they can dynamically adapt to a changing allocation of resources. The batch system can expand or shrink a running malleable job to improve system utilization, throughput, and response times. In the past, however the rigid nature of commonly used programming models like MPI made writing malleable applications a daunting task, which is why it remains largely unrealized. This is now changing. To improve fault tolerance, load imbalance, and energy efficiency in emerging exascale systems more adaptive programming paradigms are being investigated. Although they may offer better support for malleability, current batch systems still lack management facilities for malleable jobs and are therefore incapable of leveraging their potential. In this guided research we propose an extension to the SLURM resource manager to support malleable jobs.

2.1 Job Classification

As defined by Feitelson and Rudolph [1], jobs can be classified into four categories based on their flexibility.

- ***Rigid job:*** This is the most common type which requires a fixed number of processors throughout its execution.
- ***Moldable job:*** In this kind of job the resource set can be molded or modified by the batch system before starting the job (e.g. to effectively fit alongside other rigid jobs). Once started its resource set cannot be changed anymore.
- ***Evolving job:*** These kind of jobs request for resource expansion or shrinkage during their execution. Applications that use multiscale analysis like Quadflow or Adaptive mesh refinement(AMR) exhibit this kind of behavior typically due to unexpected increases in computations or having reached hardware limits(e.g. memory) on a node.
- ***Malleable job:*** The expansion and shrinkage of resources are initiated by the batch system in contrast to the evolving jobs. The application adapts itself to the changing resource set.

The first two types fall into the category of what is called as the static allocation since the allocation of rigid and moldable jobs must be finalized before the job starts. Whereas, the last two types fall under the category of dynamic allocation since this property of expanding or shrinking evolving and malleable jobs(together termed adaptive jobs) happens at runtime.

Malleable jobs hold a strong potential to obtain high system performance. Batch systems can substantially improve the system utilization, throughput and response times with efficient shrink/expand strategies for malleable jobs. Similarly, applications also profit when expanded with additional resources as this can increase application speedup and improve load balance across the jobs resource set. Enabling malleable jobs in cluster systems requires three major components: (i) a parallel runtime that is able to adapt to a changing resource set, (ii) a batch system with dynamic allocation facilities, and (iii) a communication mechanism between the two. Traditionally, all batch systems support only static allocations.

2.2 Traditional Resource Management

The role of a resource manager is that it acts like a *glue* for a parallel computer to execute parallel jobs. It should make a parallel computer as easy to use as almost a PC. MPI would typically be used to manage communications within the parallel program. A resource manager allocates resources within a cluster, launches and otherwise manages jobs. Some of the examples of widely used open source as well as commercial resource managers are **SLURM, TORQUE, OMEGA, IBM Platform LSF** etc. Together with a scheduler it is termed as a ***Batch System***. The Batch System serves as a middleware for managing supercomputing resources. The combination of *Scheduler+Resource Manager* makes it possible to run parallel jobs.

The role of a job scheduler is to manage queue(s) of work when there is more work than resources. It supports complex scheduling algorithms which are optimized for network topology, energy efficiency, fair share scheduling, advanced reservations, preemption, gang scheduling(time-slicing jobs) etc. It also supports resource limits(by queue, user, group, etc.). Many batch systems provide both resource management and job scheduling within a single product (e.g. LSF) while others use distinct products(e.g. Torque resource manager and Moab job scheduler). Some other examples of Job scheduling systems are **LoadLeveler, OAR, Maui, SLURM** etc.

The prime focus of this work will be on SLURM(Simple Linux Utility For Resource Management) which will be the choice of batch system upon which the support for Invasive Computing will be demonstrated. SLURM is a sophisticated open source batch system with about 500,000 lines of C code whose development started in the year 2002 at Lawrence Livermore National Laboratory as a simple resource manager for Linux Clusters and a few years ago spawned into an independent firm under the name SchedMD. SLURM has since its inception also evolved into a very capable job scheduler through the use of optional plugins. It is used on many of the world's largest supercomputers and is used by a large fraction of the world's TOP500 Supercomputer list. It supports many UNIX falvors like AIX, Linux, Solaris and is also fault tolerant, highly scalable, and portable.

Plugins are dynamically linked objects loaded at run time based upon configuration file and/or user options. Approximately 80 plugins of different varities are currently available. Some of them are listed below:

- ***Accounting storage:*** MySQL, PostgreSQL, textfile.
- ***Network Topology:*** 3D-Torus, tree.

- ***MPI:*** OpenMPI, MPICH1, MVAPICH, MPICH2, etc.

PLugins are typically loaded when the daemon or command starts and persist indefinitely. They provide a level of indirection to a configurable underlying function.

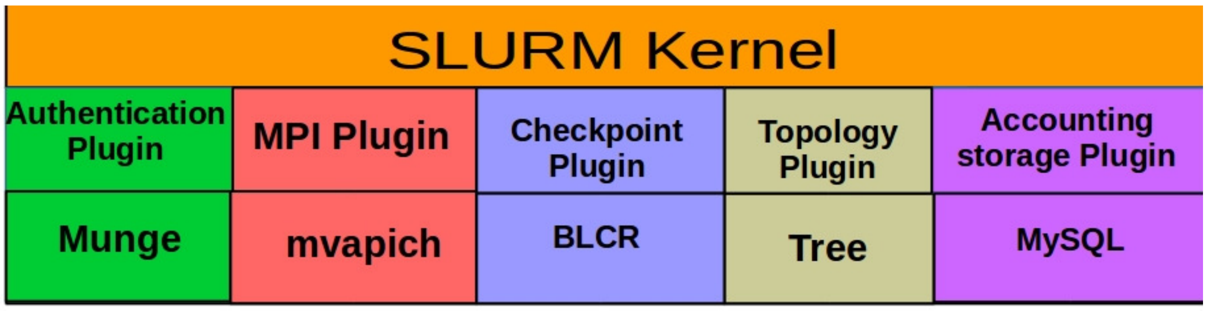


Figure 1: Software layers of SLURM

2.3 Support For Invasive Computing

Existing batch systems usually only support static allocation of resources to applications before job start. Hence we need some kind of an Invasive Resource Management to be integrated into these existing batch systems so that we can support malleable jobs allowing us to change the allocated resources dynamically at runtime. In order to achieve this we will follow the below approach:

- ***Invasive Resource Manager (alias iHypervisor):*** An independent component which will talk to the current batch systems via a communication mechanism(protocol) to obtain invasive job(s) submitted specifically to the invasic partition that can support invasive computing. The iHypervisor will then take these jobs and perform some kind of runtime scheduling for pinning these jobs to the resources in the partition and makes these decisions in order to optimize certain local metrics such as resource utilization, power stability, energy efficiency etc. The scheduling here is done at the granularity of cores and sockets. iHypervisor is the one that has the complete information of the resources in the invasic partition and also manages them. This component in an independent entity with the purpose of inter-operating with existing batch systems rather than replacing them with an entirely new one. It may be possible that in the future this component will not be a separate entitiy but will be built into the batch system itself.

- ***Invasive Job Scheduler (alias iScheduler):*** This component will be a new extension built into the existing batch systems for performing job scheduling. The scheduling decisions are communicated via the protocol used to speak to iHypervisor and these decisions are basically job(s) selected via a scheduling algorithm to be submitted to the iHypervisor for execution. The scheduling decisions will be made on the basis of available resources in the partition and it is the iHypervisor that communicates this to iScheduler in the form of resource offers (Real/Virtual). It can be a virtual resource offer because the iHypervisor can hide the real resources and present a rather fake view of them to iScheduler in the hope of getting a mapping of jobs to offer that is more suitable to satisfy its local metrics. Similar to iHypervisor, the iScheduler makes its decisions to optimize for certain local metrics such as high job throughput, reduced job waiting times, deadlines, priorities etc. This highlights the mismatching policies/metrics for which both the iHypervisor and iScheduler make their decisions on and hence both will be involved in some kind of a negotiation via the protocol to reach a common agreement.

- ***Negotiation Protocol:*** This protocol forms the core of the interaction between the iScheduler and iHypervisor. It allows for iHypervisor to make one or a set of resource offers to iScheduler which then needs to select jobs from its job queue to be mapped to these resource offers and finally sent back to the iHypervisor. The iHypervisor will then decide whether to accept/reject this mapping to satisfy its local metrics. If it accepts it will launch them based on some run time scheduling and if it rejects then it informs this to iScheduler in addition to sending it a new resource offer. The iScheduler can also reject the resource offers in which case it will be sent a new one. On accepting an offer, the iScheduler then repeats its tasks to send back a mapping to iHypervisor and this interaction continues until both reach a common agreement. If the number of such attempts reach a threshold then iScheduler will just accept whatever offer it receives and iHypervisor will also accept whatever Map:Jobs$\rightarrow$Offers it receives closing this transaction of negotiating. After this a new transaction will start.

- ***SLURM:*** SLURM is our choice of an existing batch system upon which this new implementation will be demonstrated as a proof of concept to support the new paradigm of resource-aware programming in the domain of invasive computing.

This project implements a testing prototype for demonstrating how such an approach to support invasive computing with the above entities may work. It will involve implementing the communication infrastructure using SLURM API due to which iHypervisor and iScheduler will interact with each other using protocol messages filled with dummy values. It will also involve implementing the iScheduler as a new plugin(multithreaded) for SLURM and iHypervisor as a fake multithreaded Invasive Resource Manager daemon with bare minimum functionalities that in the near future will be an enhanced version of the daemon slurmd(built on top of it) found in SLURM. For the purposes of testing this prototype, the different scenarios that can be observed most of the time with the kind of negotiation protocol described earlier would be verified.

3 Design and Implementation

This section illustrates and describes a high level design of the software implemented with the help of protocol sequence diagrams and state machine diagrams. It will help to understand at a high level as to how the system has been designed to support this new approach of invasive computing and how will many of its components in the software hierarchy interact with each other with new protocols or extensions of existing protocols to integrate such an invasive resource management into current batch systems.

3.1 Software Architecture

The following page shows the software architecture of how Invasive Resource Management can be supported with a traditional resource manager and how exactly the new software components will fit in the existing software hierarchy. The figure [X] relates closely to how SLURM is organized since the intention of this work would be to demonstrate the support for Invasive Computing with the help of SLURM as a resource manager.

- The top layer is that of the core resource management component which has access to job queues. In this architecture, it will now have access to not only the queue for the legacy(static) jobs but also invasive job queue(jobs submittted to invasic partition that supports invasive computing).
- In a traditional setup the top layer will perform the task of job scheduling as well. This means that it will select a job(s) from the queue of jobs based on the current state of resources and many other factors to dispatch it to the traditional process manager below in the hierarchy. The process manager then takes the responsibility of launching these jobs on the allocated resources in the partition and managing them for their full lifetime. In case of parallel jobs, it will manage the job in a parallel environment along with facilitating the communication amongst the parallel tasks/processes with the help of a PMI(Process Manager Interface) server. The process manager may also spawn slave daemons on each of the nodes which are a part of the resource allocation for a single job to manage them more effectively.
- As discussed in the previous chapter, an independent Invasive resource management component by the name "iHypervisor" will be implemented which needs to communicate with a new scheduling component iScheduler and influence the scheduling decisions taken by it. The iHypervisor sits between the top layer and the process manager.
- A new job scheduler specifically for invasive jobs needs to be integrated into the existing batch system. This is due to the reason that the scheduler for invasive jobs will work in a different manner based on the approach described earlier in comparison to the legacy job scheduler for static jobs. In case of SLURM which has a modular design with several optional plugins, a new plugin by name "iScheduler" will be implemented for SLURM to handle job scheduling specifically for invasic jobs.

- Communication between iHypervisor and iScheduler will involve the negotiation protocol as explained in the previous chapter but will also include periodic feedbacks being sent by iHypervisor to iScheduler having some useful statistical measures about current state of resources, resource utilization, job throughput etc. that may help influence the decision making of iScheduler. This communication will also additionally support a means to service urgent jobs immediately.

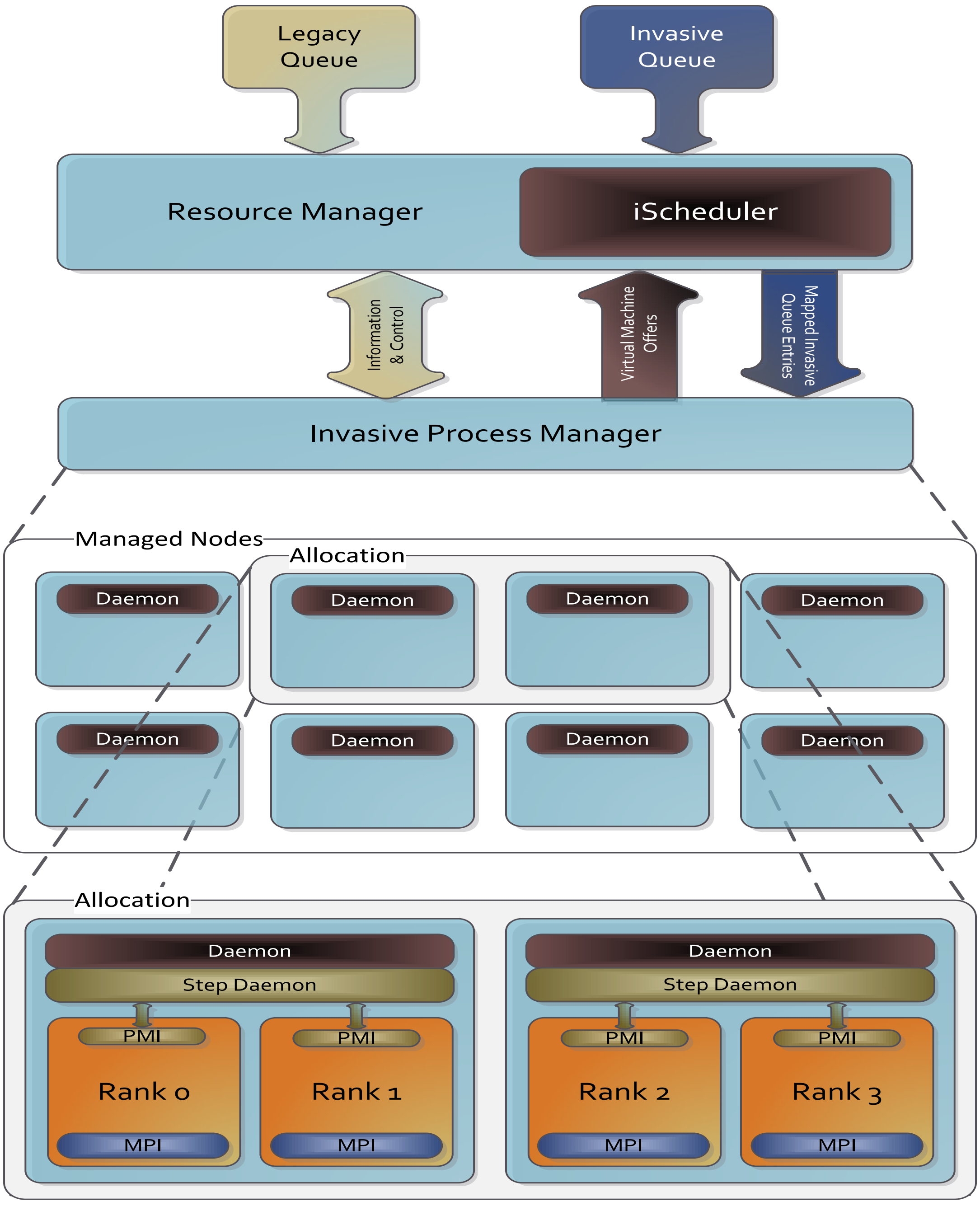
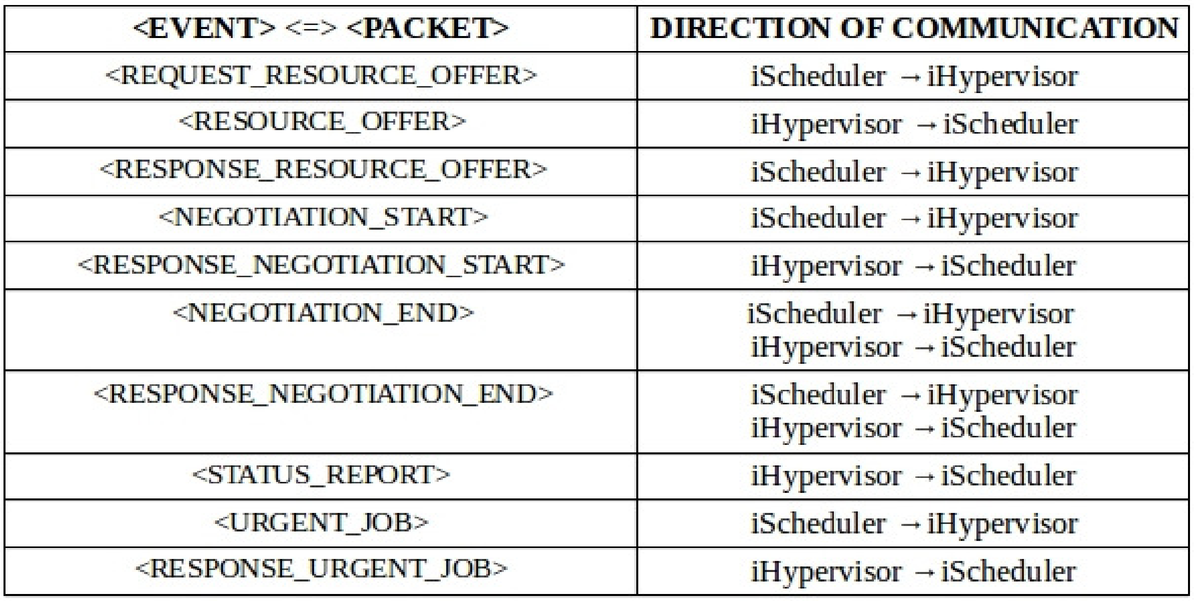


Figure 2: Invasive Resource Management Architecture



<EVENT> <=> <PACKET>	DIRECTION OF COMMUNICATION
<REQUEST_RESOURCE_OFFER>	iScheduler → iHypervisor
<RESOURCE_OFFER>	iHypervisor → iScheduler
<RESPONSE_RESOURCE_OFFER>	iScheduler → iHypervisor
<NEGOTIATION_START>	iScheduler → iHypervisor
<RESPONSE_NEGOTIATION_START>	iHypervisor → iScheduler
<NEGOTIATION_END>	iScheduler → iHypervisor iHypervisor → iScheduler
<RESPONSE_NEGOTIATION_END>	iScheduler → iHypervisor iHypervisor → iScheduler
<STATUS_REPORT>	iHypervisor → iScheduler
<URGENT_JOB>	iScheduler → iHypervisor
<RESPONSE_URGENT_JOB>	iHypervisor → iScheduler

Figure 3: Message Types

3.2 Communication Phases

- ***Protocol Initialization:*** This phase basically establishes the initial environment between the communicating parties (iScheduler and iHypervisor) for proper communication later on. Successful initialization of this phase prepares both the parties to start negotiating based on the negotiation protocol described in the following points. During this protocol initialization various parameters such as protocl version, maximum attempts for negotiation, timer intervals and several others could be exchanged to set up the internal data structures and configuration tables for both the communicating parties. This protocol is a bi-directional communication.

- ***Protocol Finalization:*** This phases signals the end of communication between iHypervisor and iScheduler using negotiation protocol. It leads to a safe termination of this communication followed by the release of any internal data structures allocated earlier along with configuration parameters. This results in consistent behaviour of both the communicating parties which can then proceed to safely terminate and exit. This protocol is a bi-directional communication.

- ***Negotiation:*** This is the most important phase in this whole approach to support invasive computing as discussed in the previous chapter. It is the phase during which both iHypervisor and iScheduler are negotiating with each other till they reach an agreement. If they do not then they continue till a certain limit to the number of negotiating attempts are reached after which both of them just agree in their final attempt closing the current negotiation. After this a new transaction of negotiation begins.

- ***Feedback:*** This concerns the periodic feedback sent by the iHypervisor to the iScheduler containing useful information such as the job states, latest snapshot of the resources in the invasic partition and many other statistical measures not limited to system utilization, job throughput, waiting times of jobs to help and influence the iScheduler in its decision making for scheduling jobs during its future transactions of negotiation. This protocol is a uni-directional communication.

- ***Urgent Jobs:*** This protocol concerns the support for urgent jobs. At any given point of time a cluster or supercomputing center may want to support very high priority jobs immediately without any further delay. By introducing support for invasive computing, it makes it all the more feasible to help run these urgent jobs immediately by either shrinking the resources of other jobs or suspending/Killing them.

The following pages illustrate some sequence diagrams of how the negotiation protocol works with some of the important scenarios that will be encountered most of the time during the operation of the system. The sequence diagrams of other protocols are not shown in this document since they are simple and commonly observed type of protocols.

3.3 Protocol Sequence Diagrams

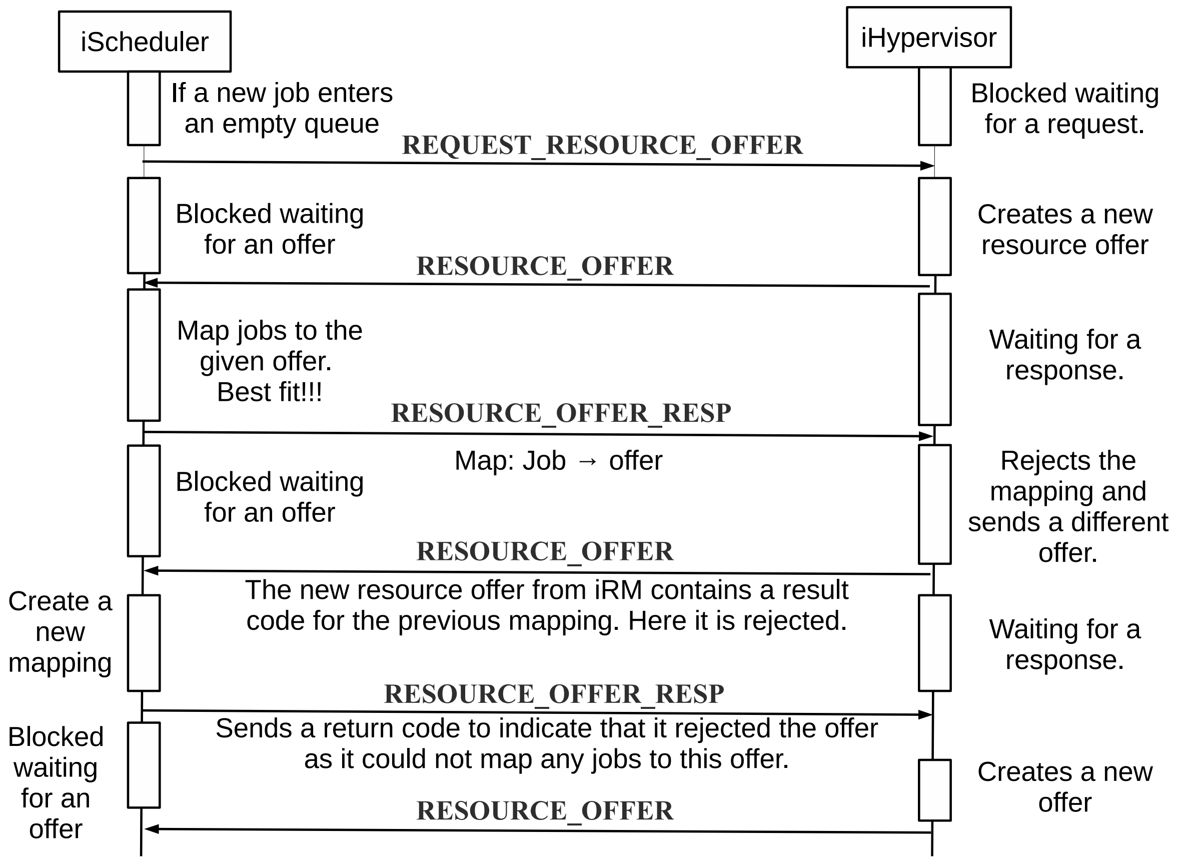

iScheduler
iHypervisor
If a new job enters an empty queue
Blocked waiting for a request.
REQUEST_RESOURCE_OFFER
Blocked waiting for an offer
Creates a new resource offer
RESOURCE_OFFER
Map jobs to the given offer. Best fit!!!
Waiting for a response.
RESOURCE_OFFER_RESP
Map: Job → offer
Blocked waiting for an offer
Rejects the mapping and sends a different offer.
RESOURCE_OFFER
The new resource offer from iRM contains a result code for the previous mapping. Here it is rejected.
Create a new mapping
Waiting for a response.
RESOURCE_OFFER_RESP
Sends a return code to indicate that it rejected the offer as it could not map any jobs to this offer.
Blocked waiting for an offer
Creates a new offer
RESOURCE_OFFER

Figure 4: Scenario 1

- Above diagram illustrates a scenario where both iScheduler and iHpervisor are negotiating with each other. The scenario is continued in the next page. Figure[X] illustrates another scenario where negotiations may stop when job queue becomes empty and iHypevisor then will wait for a request from iScheduler for a resource offer that will happen when new jobs arrive.

- iScheduler makes scheduling decisions at a coarser level of granularity which is nodes whereas iHypervisor does at the granularity of cores and sockets. Both will negotiate with each other till they reach an agreement.

- It is an event based scheduling which means iScheduler makes a scheduling decision only when it is triggered by receiving a resource offer from iHypervisor. It is only at the start when there are no jobs in the queue and during the operations when the queue may become empty that the iScheduler will have to explicitly send a request message to iHypervisor for a resource offer otherwise at all other times scheduling is event based.

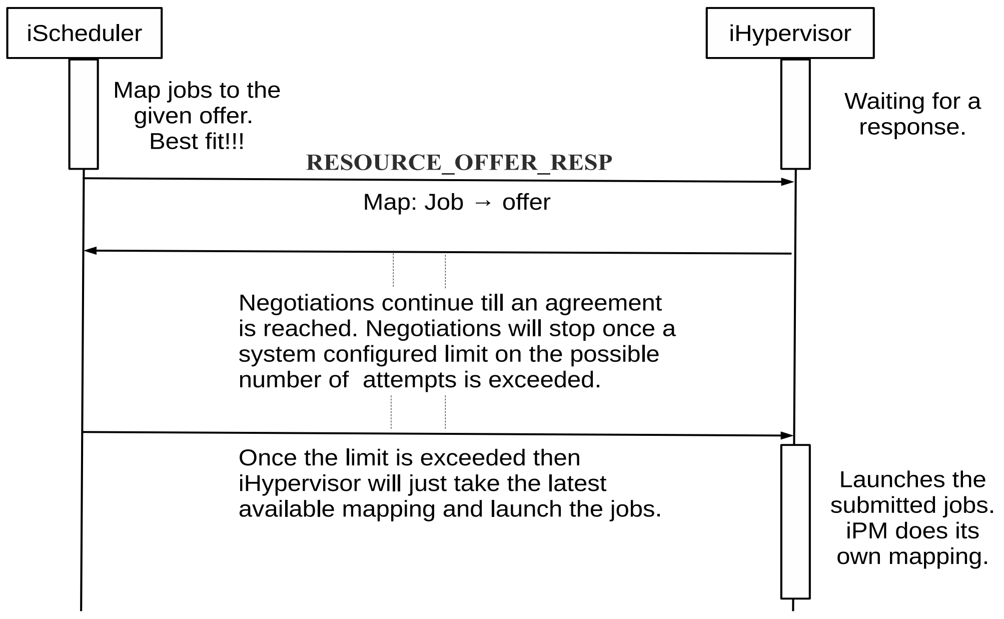

iScheduler
iHypervisor
Map jobs to the given offer. Best fit!!!
Waiting for a response.
RESOURCE_OFFER_RESP
Map: Job → offer
Negotiations continue till an agreement is reached. Negotiations will stop once a system configured limit on the possible number of attempts is exceeded.
Once the limit is exceeded then iHypervisor will just take the latest available mapping and launch the jobs.
Launches the submitted jobs. iPM does its own mapping.

Figure 5: Scenario 1 contd.

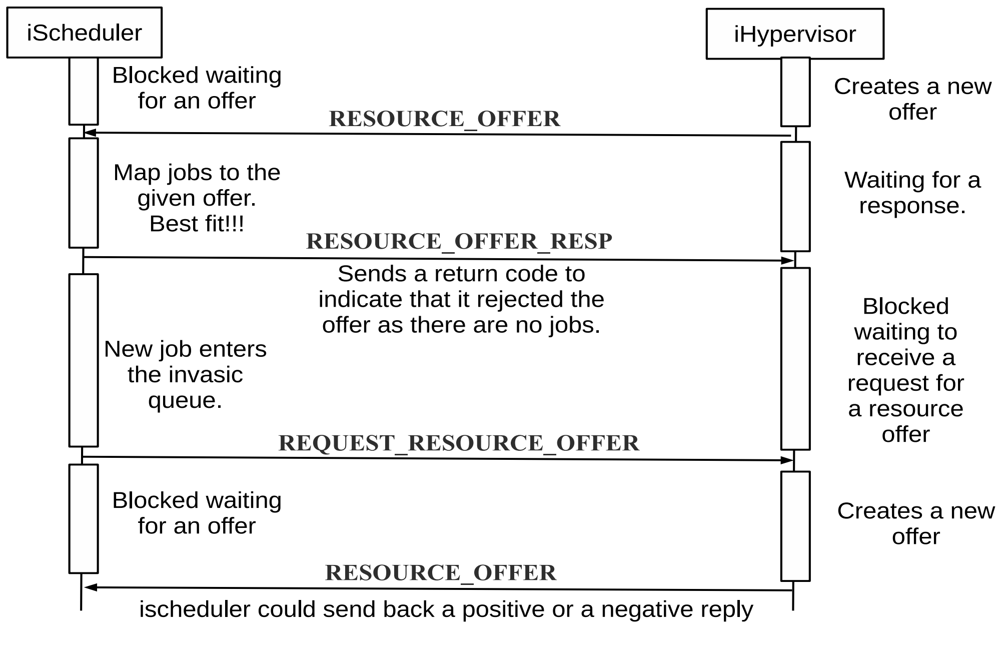
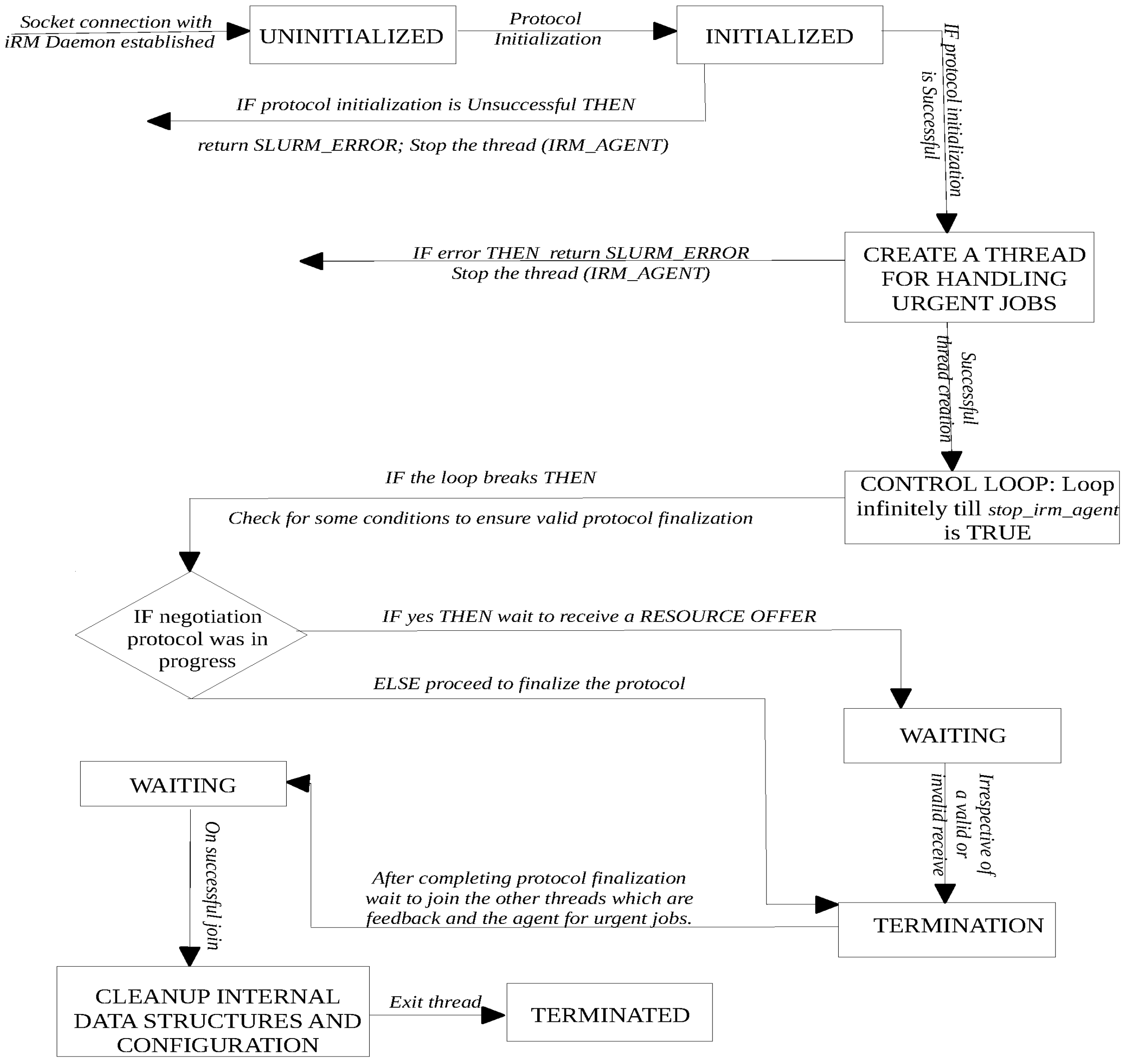

iScheduler
iHypervisor
Blocked waiting for an offer
Creates a new offer
RESOURCE_OFFER
Map jobs to the given offer. Best fit!!!
Waiting for a response.
RESOURCE_OFFER_RESP
Sends a return code to indicate that it rejected the offer as there are no jobs.
New job enters the invasic queue.
Blocked waiting to receive a request for a resource offer
REQUEST_RESOURCE_OFFER
Blocked waiting for an offer
Creates a new offer
RESOURCE_OFFER
ischeduler could send back a positive or a negative reply

Figure 6: Scenario 2

3.4 State Machine Diagrams

This section focuses on iScheduler and a thread iRM_AGENT that it spawns which is the one responsible for all the communication with the iHypervisor including spawning other agent threads for handling feedbacks and urgent jobs.



Socket connection with iRM Daemon established
UNINITIALIZED
Protocol Initialization
INITIALIZED
IF protocol initialization is Successful
IF protocol initialization is Unsuccessful THEN
return SLURM_ERROR; Stop the thread (IRM_AGENT)
IF error THEN return SLURM_ERROR
Stop the thread (IRM_AGENT)
CREATE A THREAD FOR HANDLING URGENT JOBS
Successful thread creation
CONTROL LOOP: Loop infinitely till stop_irm_agent is TRUE
IF the loop breaks THEN
Check for some conditions to ensure valid protocol finalization
IF negotiation protocol was in progress
IF yes THEN wait to receive a RESOURCE OFFER
ELSE proceed to finalize the protocol
WAITING
Irrespective of a valid or invalid receive
WAITING
On successful join
After completing protocol finalization wait to join the other threads which are feedback and the agent for urgent jobs.
TERMINATION
CLEANUP INTERNAL DATA STRUCTURES AND CONFIGURATION
Exit thread
TERMINATED

Figure 7: iRM Agent

- Above diagram and the ones in the following pages illustrate state machine diagrams for few of the communication phases described earlier starting first with a general diagram of how the mulithreaded component iRM agent inside iScheduler starts up and shuts down.

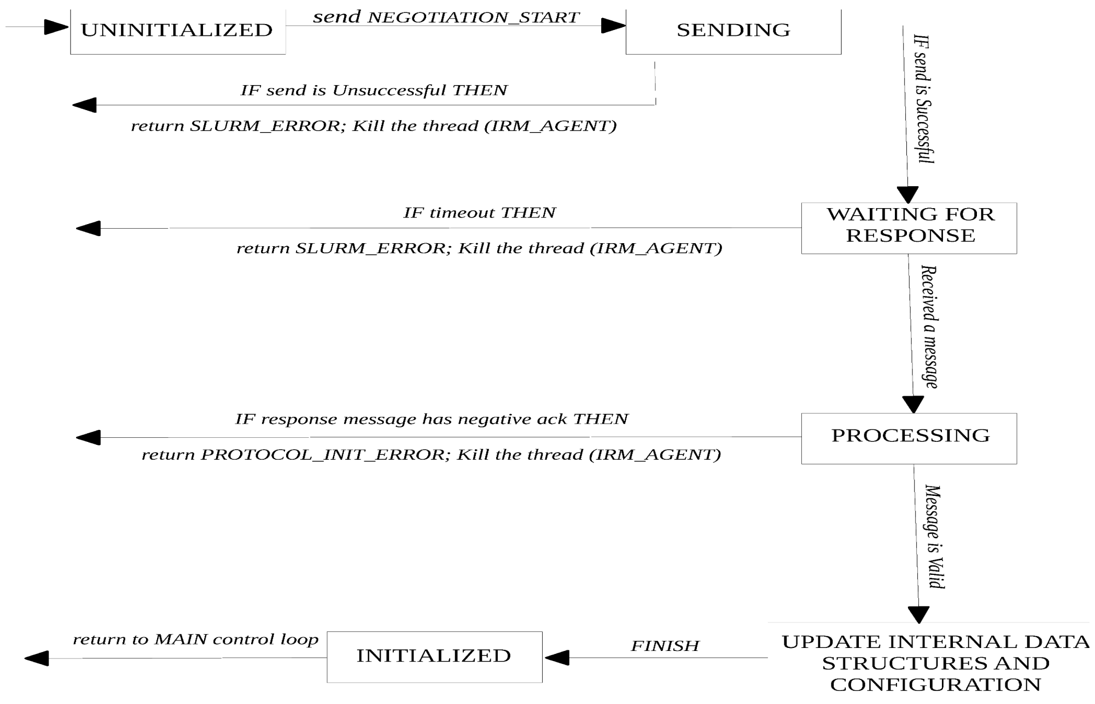

UNINITIALIZED
send NEGOTIATION_START
SENDING
IF send is Unsuccessful THEN
return SLURM_ERROR; Kill the thread (IRM_AGENT)
IF send is Successful
WAITING FOR RESPONSE
IF timeout THEN
return SLURM_ERROR; Kill the thread (IRM_AGENT)
Received a message
PROCESSING
IF response message has negative ack THEN
return PROTOCOL_INIT_ERROR; Kill the thread (IRM_AGENT)
Message is Valid
UPDATE INTERNAL DATA STRUCTURES AND CONFIGURATION
FINISH
INITIALIZED
return to MAIN control loop

Figure 8: Protocol Initialization

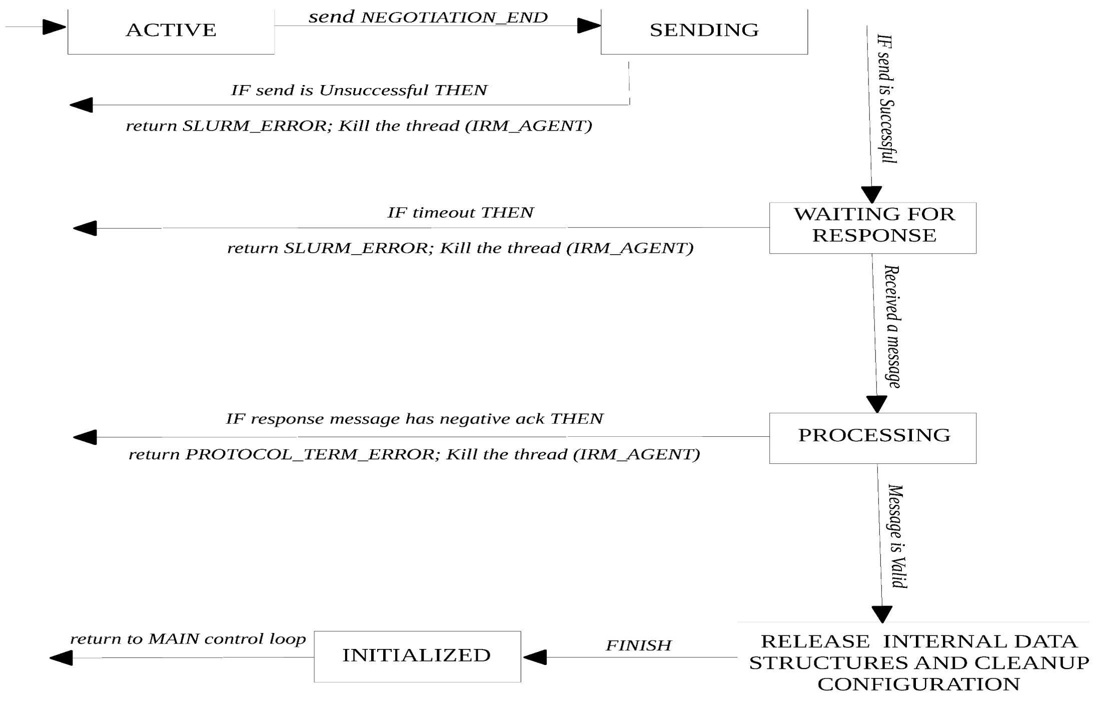

ACTIVE
send NEGOTIATION_END
SENDING
IF send is Unsuccessful THEN
return SLURM_ERROR; Kill the thread (IRM_AGENT)
IF send is Successful
WAITING FOR RESPONSE
IF timeout THEN
return SLURM_ERROR; Kill the thread (IRM_AGENT)
Received a message
PROCESSING
IF response message has negative ack THEN
return PROTOCOL_TERM_ERROR; Kill the thread (IRM_AGENT)
Message is Valid
RELEASE INTERNAL DATA STRUCTURES AND CLEANUP CONFIGURATION
FINISH
INITIALIZED
return to MAIN control loop

Figure 9: Protocol Termination

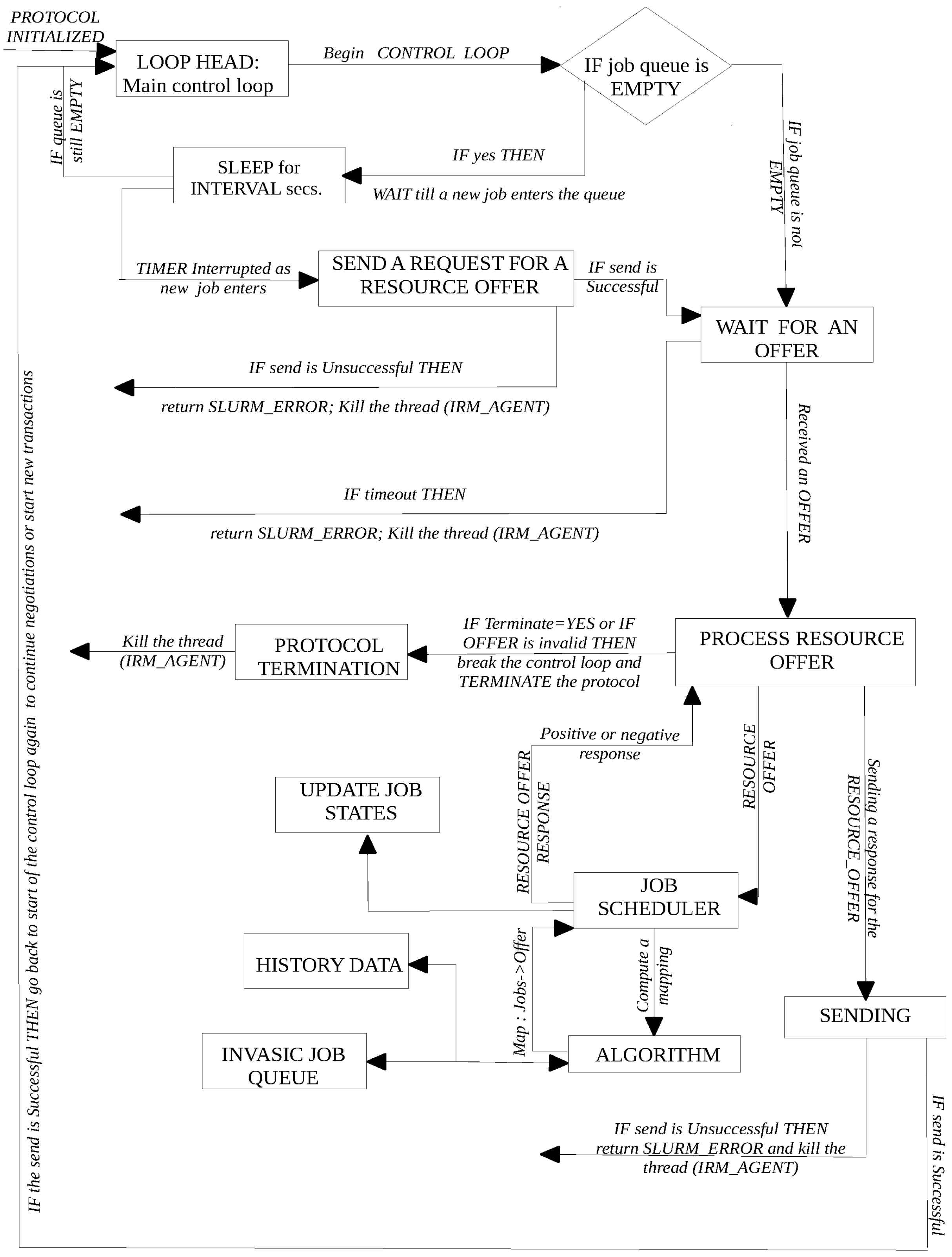

PROTOCOL INITIALIZED
LOOP HEAD: Main control loop
Begin CONTROL LOOP
IF job queue is EMPTY
IF queue is still EMPTY
IF yes THEN
WAIT till a new job enters the queue
SLEEP for INTERVAL secs.
IF job queue is not EMPTY
TIMER Interrupted as new job enters
SEND A REQUEST FOR A RESOURCE OFFER
IF send is Successful
WAIT FOR AN OFFER
IF send is Unsuccessful THEN
return SLURM_ERROR; Kill the thread (IRM_AGENT)
Received an OFFER
IF timeout THEN
return SLURM_ERROR; Kill the thread (IRM_AGENT)
IF the send is Successful THEN go back to start of the control loop again to continue negotiations or start new transactions
Kill the thread (IRM_AGENT)
PROTOCOL TERMINATION
IF Terminate=YES or IF OFFER is invalid THEN break the control loop and TERMINATE the protocol
PROCESS RESOURCE OFFER
Positive or negative response
RESOURCE OFFER RESPONSE
RESOURCE OFFER
Sending a response for the RESOURCE_OFFER
UPDATE JOB STATES
JOB SCHEDULER
HISTORY DATA
Map : Jobs->Offer
Compute a mapping
SENDING
INVASIC JOB QUEUE
ALGORITHM
IF send is Unsuccessful THEN return SLURM_ERROR and kill the thread (IRM_AGENT)
IF send is Successful

Figure 10: Negotiation

4 Conclusion

The objective of this guided research was to implement a basic prototype to serve as a proof of concept for supporting a new programming paradigm called resource-aware programming in the domain of invasive computing for the upcoming exascale era. This will serve as a basic groundwork for further concrete implementations of scheduling algorithms, run time tuning of parallel applications, optimizing decision making at both the levels of iScheduler and iHypervisor through different approaches one of which could be machine learning. The basic communication infrastructure including the skeleton of protocol messages and the respective multithreaded communicating parties have been successfully implemented and tested for correctness.

References