Software Design Document: GPU N-Body Integration Toolkit

Prepared by:
Daniel Bagnell
Jason Economou
Rajkumar Jayachandran
Tim McJilton
Gabe Schwartz
Andrew Sherman

Advisor: Prof. Jeremy Johnson

Stakeholders: Prof. Steve McMillan Alfred Whitehead

Contents

1	Intro	oduction 4
	1.1	Purpose
	1.2	Scope
	1.3	Glossary
2	Arcl	nitecture 5
3	Gra	phical User Interface 5
	3.1	Overview
	3.2	Network Interaction
	3.3	Node Management
4	Exp	eriment Components 8
	4.1	Experiment Object Model
		4.1.1 Experiment Class
	4.2	Particles
		4.2.1 Particle Class
	4.3	Modules
		4.3.1 Module Class
		4.3.2 Parameter Class
		4.3.3 Unit Class
		4.3.4 CompoundUnit Class
		4.3.5 UnitType Enumeration
		4.3.6 SIPrefix Enumeration
		4.3.7 AstrophysicalDomain Enumeration
		4.3.8 StoppingConditions Enumeration
	4.4	Diagnostics
		4.4.1 Object Model
		4.4.2 Diagnostic Class
		4.4.3 Experiment Manager Class
		4.4.4 Condition Class
	4.5	Logging
		4.5.1 Object Model
		4.5.2 Logging Class
5	Con	imand Line Tool 22
	5.1	Overview
	5.2	Using the CLT
		5.2.1 CLT Flags
	EO	23. Dunning The Evansiment
	5.3	Running The Experiment
		5.3.1 Simulation Overview

6	Net	working		26				
	6.1	Overvi	ew	. 26				
	6.2	Netwo	rk Software Components	. 26				
		6.2.1	Control Instance	. 26				
		6.2.2	Node Instance	. 27				
	6.3	Object	: Model	. 27				
		6.3.1	Overview	. 27				
		6.3.2	Control Instance					
		6.3.3	Node Instance					
		6.3.4	TransmissionThread	. 28				
		6.3.5	Receiving Thread	. 28				
		6.3.6	ListeningThread					
		6.3.7	Packet Objects					
7	Data	а		30				
•	7.1		ment Specification					
		7.1.1	Experiment					
		7.1.2	Time					
		7.1.3	Module					
		7.1.4	Param					
		7.1.5	Particle					
		7.1.6	Diagnostic					
		7.1.7	Logger					
		7.1.8	Stopping Condition	_				
	7.2		rk Packets					
	1.2	7.2.1	Packet Header					
		7.2.1	Status Query Packet					
		7.2.3	Status Response Packet					
		7.2.4						
		7.2.4	Capability Query Packet					
		1.2.5	Capability Response Facket	. 33				
8	Appendix 33							
	8.1	1 Requirements Traceability Table						

1 Introduction

1.1 Purpose

This document serves as the system design for the GPUnit framework to control AMUSE experiments as set forth in the Software Requirements Specification. AMUSE (Astrophysical Multipurpose Software Environment) is a software package and set of legacy codes for performing astrophysics simulations. For more details on the purpose and use cases of AMUSE, please refer to our SRS. The goal of this document is to provide details guiding the construction of the framework, as well as to provide a reference to the framework's architecture. It contains specifications for the objects used by the system and our initial reference implementation plans.

1.2 Scope

This document will describe the implementation for our framework, covering the network layer, the user interface and the core python command line scripts. It will cover user interface support classes as they will need to be coded, however the actual UI components such as windows and widgets will not be described. Code and objects for these items are generated as part of the Qt Creator¹ GUI development kit. As such, the code is not intended to be human-readable and is subject to change between Qt versions. Our SRS document contains the reference prototype which will serve as the design for the GUI widgets. The data models including experiment file layouts and network packet details will also be covered.

1.3 Glossary

For terms not in this glossary, please refer to the SRS.

Cluster A group of networked computing nodes available to perform some task.

Computer Node A computer that is a member of a cluster.

IPC Channel Any object or memory space used for Inter-Process Communication (IPC). Examples include UNIX Pipes, BSD/Winsock Sockets and UNIX Shared Memory (SHM).

Node Health Statistics Information about the current state of a node including properties such as CPU usage and memory usage.

 $^{^{1}} http://qt.nokia.com/products/developer-tools/\\$

2 Architecture

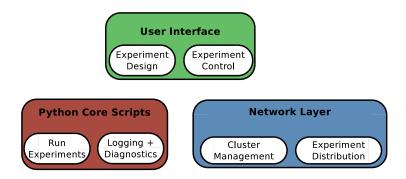


Figure 2.1: Architecture Diagram

3 Graphical User Interface

3.1 Overview

The user interface will allow the user to create and manage experiments as described in the SRS. Much of the interface code is generated from the GUI development tools, however there are some support classes required for the GUI to interact with the network and the command line tools.

3.2 Network Interaction

The network interaction classes allow the UI to send and receive messages through a control instance (Section 6.2.1). The control instance can be either running locally or on a remote machine. The UI uses this link to query the cluster for status updates and control running experiments.

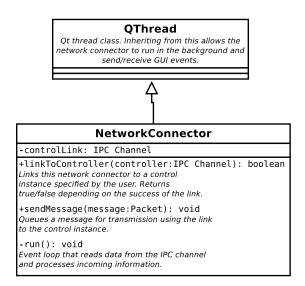


Figure 3.1: GUI Network Connection Class

3.3 Node Management

Nodes in the cluster, detected through the network connector, may be viewed in the node view window. (Satisfies requirements 2.8.1.X)

From the node window, the user can assign experiments to individual nodes. (Satisfies requirement 2.8.2.1)

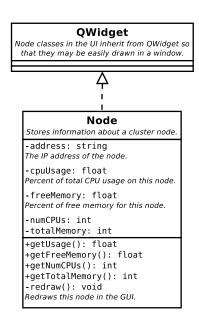


Figure 3.2: Node Class

4 Experiment Components

Experiment

4.1 Experiment Object Model

```
An experiment holds all used particles, modules,
                                                                     Sets the name of the experiment to the given
                logging tools, diagnostics, and initial
                    conditions for a simulation.
                                                                     +getName(): string
             It is able to evolve the state of the particle
                                                                     Returns the name of the experiment.
          system by the supplied timestep and retrieve the
            current state of the system. By default, this
                                                                     +setTimeStep(timeStep:UnitType): void
            evolution includes AMUSE's built-in stopping
                                                                     Sets the timestep to use in the simulation.
                           conditions.
          The supplied diagnostics gather information from
                                                                     +getTimeStep(): UnitType
           the simulation, and the supplied loggers report
                                                                     Returns the currently set timestep for the
               on the status of the system during the
                                                                     simulation.
                           simulation.
                                                                      +disableStopConditions(): void
          An experiment is also able to read from and write
                                                                     Disables stopping conditions by setting
               to XML files for persistence and reuse.
                                                                     stopisEnabled to faise.
name: string
                                                                      enableStopConditions(): void
The name of the experiment
                                                                     Enables stopping conditions by setting
-timeStep: UnitType
                                                                     stopisEnabled to true.
The amount of time between each snapshot of the
                                                                      ⊦addModule(module:Module): void
particle system.
                                                                     Adds a module to the experiment.
-stopIsEnabled: boolean = true
                                                                     +removeModule(module:Module): void
True if stopping conditions are enabled.
                                                                     Removes the given module from the experiment
-modules: List∢Module>
                                                                      -addModules(modules:List⊲Module>): void
A list of modules used in the experiment.
                                                                     Adds the given list of modules to the experiment.
-particles: List<Particle>
                                                                     +addParticle(body:Particle): void
The list of particles to use in the experiment.
-diagnostics: List<Diagnostic>
                                                                     +removeParticle(body:Particle): void
The list of diagnostics to use in this
                                                                     Removes the given particle from the experiment.
experiment.
                                                                     +addParticles(bodies:List<Particle>): void
-loggers: List<Logger>
                                                                     Adds the given particles to the experiment.
This list of logging tools to use in the
experiment.
                                                                      -addDiagnostic(diag:Diagnostic): void
                                                                     Adds the given diagnostic to the experiment.
+writeXMLFile(fileName:string=name): void
Writes the experiment description to the
                                                                     +removeDiagnostic(diag:Diagnostic): void
designated file in XML format
                                                                      The diagnostic to remove from the experiment
+<<static>> loadXMLFile(fileName:string): Experiment
                                                                      +addDiagnostics(diags:List<Diagnostic>): void
Loads experiment attributes from the given XML
                                                                     Adds the given list of diagnostics to the
+getCurrentState(): List<Particle>
                                                                     +addLogger(logger:Logger): void
Retrieves the list of particles to read their
                                                                     Adds the given logger to the experiment.
respective current states.
                                                                     +removeLogger(logger:Logger): void
+evolveState(): void
                                                                     Removes the given logger from the experiment.
Evolves/Updates the current state of the
                                                                      +addLoggers(loggers:List<Logger>): void
particles in the experiment by the experiment's timestep.
                                                                     Adds the given list of loggers to the experiment.
```

+setName(name:string): void

Figure 4.1: Experiment Class Diagram

4.1.1 Experiment Class

An instance of the Experiment class represents a single AMUSE experiment. It holds all particles (Satisfies requirement 2.3.3), modules, logging tools, diagnostics, and initial conditions for a simulation. These are described in more detail below. The experiment class can be viewed in Figure 4.1.

4.1.1.1 Experiment Attributes Experiments have the following attributes. The type UnitType is defined later in this document.

Name	Туре	Description	
name	string	The name of the experiment.	
timeStep	UnitType	The amount of time between	
		each snapshot of the simulation.	
stopIsEnabled	boolean	Whether or not stopping	
		conditions are enabled.	
modules List <module></module>		The list of Modules used in the	
		experiment.	
particles List <particle></particle>		The list of Particles to use in the	
		experiment.	
diagnostics List <diagnostic></diagnostic>		The list of Diagnostics to use in	
		the experiment.	
loggers List <logger></logger>		The list of Loggers to use in the	
		experiment.	

4.1.1.2 Experiment Operations

void writeXMLFile(string		static Experiment loadXMLFile(string fileName)	
	eName)	Input	A string containing the filename to load the
Input	filename to write the Experiment XML file to.		Experiment XML from.
			The Experiment specified by the XML
Output (none)			file.
Description Writes the Experiment in XML format to the specified filename.		Description	Recreates an Experiment from its XML specification.

List <particle></particle>			
getCurrentState()		VO	id evolveState()
Input	(none)	Input	(none)
Output	The list of Particles in	Output	(none)
	the Experiment.	Description	Evolves/Updates the
Description	Returns the list of		current state of the
	Particles in the		particles in the
	simulation with their		Experiment by the
	updated attributes to		Experiment's timeStep.
	read their current states.		,

void setNa	me(string name)		
Input	Input A string containing the		ring getName()
	updated Experiment	Input	(none)
	name.	Output	A string containing the
Output	(none)		name of the Experiment.
Description	Updates the	Description	Returns the name of the
	Experiment's name to		Experiment.
	the given parameter.		
	neStep(UnitType		
tir	neStep)	Unit	ts getTimeStep()
Input	A UnitType containing	Input	(none)
	the updated Experiment	Output	A UnitType containing
	timeStep.		the Experiment
Output	(none)		timeStep.
Description	Updates the	Description	Returns the timeStep of
	Experiment's timeStep		the Experiment.
	to the given parameter.		
void disable	StopConditions()	void enableStopConditions()	
Input	(none)	Input	(none)
Output	(none)	Output	(none)
Description	Disables the	Description	Enables the
	Experiment's stopping		Experiment's stopping
	conditions.		conditions.
	Module(Module	void removeModule(Module module)	
п	nodule)	Input	A Module to remove
Input	A Module to add to the		from the Experiment.
	Experiment.	Output	(none)
Output	(none)	Description	Removes the given
Description	Adds the given Module		Module from the
	to the Experiment.		Experiment.
void addModules(List <module></module>		void addPart	icle(Particle body)
modules)			
Input	A list of Modules to add	Input	A Particle to add to the
	to the Experiment.	Output	Experiment. (none)
Output	(none)	Description	Adds the given Particle
Description	Adds the given Modules	Description	to the Experiment.
	to the Experiment.		to the Experiment.

void addParti	cles(List <particle></particle>	void removeParticle(Particle body)		
Ŀ	oodies)	Input	A Particle to remove	
Input	A list of Particles to add		from the Experiment.	
	to the Experiment.	Output	(none)	
Output	(none)	Description	Removes the given	
Description	Adds the given Particles		Particle from the	
	to the Experiment.		Experiment.	
void addDiag	gnostic(Diagnostic	void a	nddDiagnos-	
	diag)	tics(List	<diagnostic></diagnostic>	
Input	A Diagnostic to add to	ı	diags)	
	the Experiment.	Input	A list of Diagnostics to	
Output	(none)		add to the Experiment.	
Description	Adds the given	Output	(none)	
	Diagnostic to the	Description	Adds the given	
	Experiment. Satisfies		Diagnostics to the	
	requirement 2.6.3.{1,2}.		Experiment.	
void removeD	iagnostic(Diagnostic diag)	void addLog	ger(Logger logger)	
Input	A Diagnostic to remove	Input	A Logger to add to the	
mpat	from the Experiment.		Experiment.	
Output	(none)	Output	(none)	
Description	Removes the given	Description	Adds the given Logger	
Bescription	Diagnostic from the		to the Experiment.	
	Experiment.		Satisfies requirement	
	Experiment.		2.7.1.	
void addLogs	void addLoggers(List <logger></logger>		void removeLogger(Logger logger)	
	oggers)	Input	A Logger to remove	
Input A list of Loggers to add			from the Experiment.	
Прис	to the Experiment.	Output	(none)	
Output	(none)	Description	Removes the given	
Description	Adds the given Loggers		Logger from the	
Description	to the Experiment.		Experiment. Satisfies	
	to the Experiment.		requirement 2.7.2.	

4.2 Particles

Particles are the elements that are being updated and the experiment is being run on. These particles can be anything as little as atoms to as large as star clusters. They are made up of mass, position, and velocity. (Satisfies requirements 2.3.1, 2.3.2)

4.2.1 Particle Class

Particle
-mass: Parameter -velocity: Parameter -position: Parameter
<pre>+getVelocity(): Parameter +setVelocity(v:List<double>,unit:Unit): boolean +getPosition(): Parameter +setPosition(pos:List<double>,unit:Unit): bool +getMass(): Parameter +setPass(mass:double,unit:Unit): bool</double></double></pre>

Figure 4.2: Particle Class Diagram

Particle class will be instantiated and used for storage of particle type items on the UI of GPUnit. The Particle class in AMUSE will be used in the CLT. The particle class will be used primarily for data storage for experiment creation.

4.2.1.1 Particle Attributes

	Name Type		Description		
	mass	Parameter	The current mass of the particle.		
			Will have units.		
Ī	velocity	Parameter	The current mass of the particle.		
			Will have units.		
ĺ	position	Parameter	The current position of the		
			particle. Will have units.		

4.2.1.2 Particle Methods

Parame	ter getMass()	- bool setMass(double mass, Unit unit)	
Parameter getMass() Input (none)		Input	The new value of the
Output	A parameter with the value and units of the	Output	mass and the unit it is in. Returns true if the mass
Description	current mass. Gets the current value of	Output	is successfully set.
Description	the mass.	Description	Sets the mass of the particle.

_		bool setVelocity(List <double> v, Unit unit)</double>	
Parameter getVelocity()		Input	The new vector value of
Input	(none)		the velocity and the unit
Output	A parameter with the		it is in.
	value and units of the	Output	Returns true if the
	current velocity.		velocity is successfully
Description	Gets the current value of		set.
	the velocity.	Description	Sets the velocity of the
			particle.

		bool setPosition(List <double> pos, Unit unit)</double>	
Paramete	Parameter getPosition()		The new vector value of
Input	(none)	-	the position and the unit
Output	A parameter with the		it is in.
	value and units of the	Output	Returns true if the
	current position.		position is successfully
Description	Gets the current value of		set.
	the position.	Description	Sets the position of the
			particle.

4.3 Modules

This section defines an object model for the representation of and interaction with AMUSE modules, including several ancillary types utilized by the module implementation.

4.3.1 Module Class

Module
+name: string
+description: string
+domain: AstrophysicalDomain
+codeName: string
+codeLocation: string
+isParallel: boolean
+stoppingConditions: StoppingConditions
+parameters: List <parameter></parameter>
+toXml(): string
+fromtXml(element:string): Module

Figure 4.4: Module class diagram

An instance of the Module class represents a particular AMUSE module, providing an interface between GPUnit and the AMUSE code. The members of the Module class are used by GPUnit to properly locate and initialize an AMUSE module, as well as display a module's details and configuration to the user in the graphical interface. (Satisfies requirements 2.2.1, 2.2.2, 2.2.3, 2.2.4, 2.2.5)

4.3.1.1 Module Attributes

3.1.1 Module Attri Name	Туре	Description
name	string	The full name of the module.
description	string	A description of the module's
		purpose (in other words, the
		calculations performed by the
		module).
domain	AstrophysicalDomain	The astrophysical domain into
		which the module has been
		sorted.
codeName	string	The name of the AMUSE class
		containing the module code.
codeLocation	string	The location of the AMUSE
		module code.
isParallel	boolean	Whether the module's
		calculations can be parallelized
		across multiple workers by MPI.
stoppingConditions	StoppingConditions	The condition(s) under which
		the module may stop executing
		prematurely.
parameters	List <parameter></parameter>	The module parameters. These
		module-specific values may be
		modified prior to running the
		experiment in order to fine-tune
		the module's behavior.

4.3.1.2 Module Operations

		static Module	e fromXml(string element)
		Input	A string containing an
strin	g toXml()		XML representation of a
Input	(none)		Module.
Output	A string containing an	Output	A Module whose
	XML representation of		property values are
	the Module.		specified by the given
Description	Dumps the Module to		XML element.
	an XML element.	Description	Recreates a Module
			from its XML
			specification.

4.3.2 Parameter Class

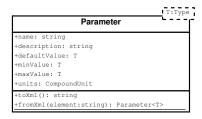


Figure 4.5: Parameter class diagram

Modules have parameters that may be modified by the user via the graphical interface. These parameters are typically specific to the module's domain or even to the individual module. The Parameter class is a generic class whose type parameter specifies the type of the parameter's value (e.g. integer or floating-point). A parameter may be a physical quantity, flag, or other value. (Satisfies Requirement 2.3.6)

4.3.2.1 Parameter Attributes

Name	Туре	Description
Т	Туре	The data type of the parameter's value.
name	string	The brief, descriptive name of the parameter.
description	string	A description of the parameter's meaning and effects.
defaultValue	Т	The parameter's default value.
minValue	Т	A lower bound on the range of valid parameter values.
maxValue	Т	An upper bound on the range of valid parameter values.
units	CompoundUnit	The physical unit(s) associated with the parameter's value.

4.3.2.2 Parameter Operations

		static Parame	ter<1> tromXml(string element)
		Input	A string containing an
S	tring toXml()		XML representation of a
Input	(none)		Parameter.
Output	A string containing an	Output	A Parameter whose
	XML representation of		property values are
the Parameter.			specified by the given
Description	Dumps the Parameter to		XML element.
an XML element.		Description	Recreates a Parameter
			from its XML
			specification.

4.3.3 Unit Class

Unit +type: UnitType +prefix: SIPrefix +exponent: integer

Figure 4.6: Unit class diagram

With AMUSE, physical quantities may be expressed using, and converted between, a number of different standard units. The Unit class, in conjunction with the CompoundUnit class, allows GPUnit to present these units to the user for the purposes of description and selection.

4.3.3.1 Unit Attributes

Name	Туре	Description
type	UnitType	The base type of astrophysical
		unit being represented.
prefix	SIPrefix	The SI prefix that modifies the
		base unit's order of magnitude.
exponent	integer	The exponent to be applied to
		the unit. For example, a typical
		unit of volume (such as cubic
		meters) would have an exponent
		of 3.

4.3.4 CompoundUnit Class

CompoundUnit
+description: string
+symbolicDescription: string
+units: List <unit></unit>

Figure 4.7: CompoundUnit class diagram

When performing physical calculations, units of measure may combined in any number of ways. The CompoundUnit class provides the mechanism by which units may be presented to the user, regardless of whether they are simple or compound.

4.3.4.1 CompoundUnit Attributes

5.4.1 Compoundome Attributes		
Name	Туре	Description
description	string	A non-abbreviated textual
		description of the combined
		physical unit.
symbolicDescription	string	A shorthand textual description
		of the combined physical unit,
		using unit abbreviations.
units	List <unit></unit>	The list of simple units whose
		combination is represented by
		this compound unit.

4.3.5 UnitType Enumeration



Figure 4.8: UnitType type diagram

UnitType enumerates the myriad of base physical units supported by AMUSE. The list of members shown may not be exhaustive.

4.3.6 SIPrefix Enumeration

<<enumeration>> **SIPrefix** +None = 0+yocto = -24+zepto = -21+atto = -18+femto = -15+nano = -9+micro = -6+milli = -3+centi = -2+deci = -1+deca = 1+hecto = 2+kilo = 3+mega = 6+giga = 9 +tera = 12+peta = 15+exa = 18+zetta = 21+yotta = 24

Figure 4.9: SIPrefix type diagram

When dealing with physical quantities, standard prefixes may be prepended to units to denote a given quantity's order of magnitude. SIPrefix enumerates the possible unit prefixes.

4.3.7 Astrophysical Domain Enumeration

Figure 4.10: AstrophysicalDomain type diagram

The modules included with AMUSE have categorized according to their domain. A module's domain indicates the quantities it deals with as well as common operations that may be called on the module. The AstrophysicalDomain type enumerates the several domains that have been specified by AMUSE.

4.3.8 StoppingConditions Enumeration

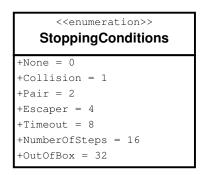


Figure 4.11: StoppingConditions type diagram

StoppingConditions is a set of bit flags enumerating the conditions under which a module may return from execution before completing its assigned calculations.

4.4 Diagnostics

Diagnostics have access to the state of the experiment at the end of each time step. Default diagnostics will be provided, e.g. printing the state to a file at a specified time step or creating a stellar mass histogram. Users can create custom diagnostic plug-ins. (Satisfies requirement 2.6)

4.4.1 Object Model

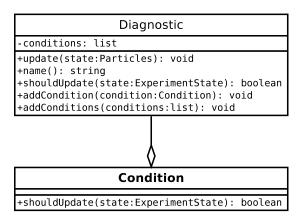


Figure 4.12: Diagnostics Object Model

4.4.2 Diagnostic Class

The Diagnostic class is an abstract class that is the base of both built-in and custom diagnostics tools. All custom diagnostic scripts must inherit from this class. Subclasses must implement the update and name operations. The Diagnostic class also contains a list of conditions to check if the diagnostic needs to be updated.

4.4.3 Experiment Manager Class

The Experiment Manager class has the responsibility to manage as well as update the active diagnostics.

4.4.4 Condition Class

The Condition class represents the conditions that need to be satisfied for a diagnostic to be updated.

4.5 Logging

Logs are helpful when the experiment must be restarted along with its output data if the experiment has an error or needs to be stopped for any other reason. Logging could be enabled or disabled by the user. The output destination for the logs could be specified. The specified destination for the output can be one among the following: the console, a file, or a window in the GUI. (Satisfies requirement 2.7)

4.5.1 Object Model

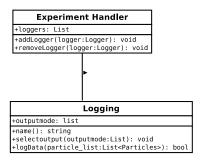


Figure 4.13: Logging Object Model

4.5.2 Logging Class

The Logging class takes care of selecting the required output option as the console, file or the GUI window.

5 Command Line Tool

5.1 Overview

The command line tool (CLT) is used for the running of an experiment file. The CLT begins with parsing the experiment file. After parsing the experiment file the CLT takes the initialization parameters and loads them using the AMUSE framework. The CLT then takes the modules specified, searches through loaded AMUSE modules and initializes the modules. The CLT links the modules together to allow interaction and updating of the current state of the system using the AMUSE channels. The CLT then loads the Logging and Data Analysis scripts specified by the experiment file, linking it to the the particle array. The CLT then runs through the time specified in the experiment file, evolving all of the modules loaded at each time step. It will run the data analysis scripts as well as the logging on the intervals specified. After the model is done evolving it will output using whatever data analysis script listed, in the file format specified.

5.2 Using the CLT

The command line tool will primarily be used by the GUI to send out jobs and run them to make the user experience much more enjoyable and at ease. The command line tool though can be used with a basic experiment file with some on the fly modifications by using some flags. (Satisfies requirement 2.4.1)

5.2.1 CLT Flags

Flag	<value being="" passed=""></value>	Description
-help	None	Displays a useful help prompt with the list of CLT flags.
-f	Experiment File Location	Loads the experiment file prompted by the filename.
-n	Number of Particles	Changes the number of particles being passed in the experiment file.
-t	End Time	Changes the max time allowed in the simulation.
-dt	Timestep	Changes the timestep in which dt moves forward.
-r	Radius	Changes the radius scaling factor.

5.3 Running The Experiment

5.3.1 Simulation Overview

A Simulation is the goal of both GPUnit and AMUSE, which makes it important to understand and break it into key parts. The Simulation can be looked at as three sections: Simulation Initialization, Running Simulation, Post Simulation. This section will go into more detail about all three of these parts and what happens in each.

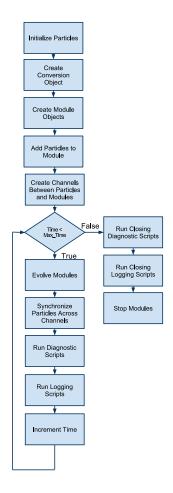


Figure 5.1: Flow Chart of Typical AMUSE Simulation

- **5.3.1.1 Simulation Initialization** Simulation Initialization is used for preparation of the particles and modules to be able to model the system. The following will break down the initialization into the main steps.
- **5.3.1.1.1 Instantiate Particles** In the initialization phase of the experiment it is important to select proper initial conditions. The main initial conditions to set is the mass distribution and and starting locations of the particles. There are built in AMUSE classes that are used to set these. The CLT while reading in the experiment file will find the initial condition files by name. (Satisfies Requirements 2.3.4)
- **5.3.1.1.2 Create Conversion Object** Most astronomical simulations use a scaling factor so you can do the numerical integration quicker. A conversion object is

a class in AMUSE that you instantiate by passing in the total mass and the average radius of the system. You will then pass this conversion object into any modules you will be using in the future so the module knows how to translate between this smaller number format to an actual real data value.

- **5.3.1.1.3 Module Creation** The CLT needs to instantiate the AMUSE class for the specific module it is working on from the location specified in the module class. It needs to instantiate the module being passed in the conversion object so the module knows what units it is working in and how to convert its units. The module then sets its list of particles to the master list of particles. After it does that it needs to run it's initial calculations needed for evolving the module.
- **5.3.1.1.4 Instantiate Channels** Channels are used to be able to communicate data to and from the modules that may be on a different system. On every time step you need to synchronize the master list of particles to every modules separate list of nodes. Afterwards you need to re-sync the modules nodes to the master node. This allows the modules to have the same data to move forward in the simulation. (Satisfies Requirement 2.5.1, 2.5.2, 2.5.2.1)
- **5.3.1.2 Running Simulation** Running the system is the area inside of the while loop. This is where the modules are run, some logging is done, some diagnostics and increment the time scale. (Satisfies Requirement 2.4.2)
- **5.3.1.2.1 Evolve Modules** The evolution of the Module is core of running a simulation. The evolution of the module is done by running the AMUSE module's evolve model function with passing in the current time.
- **5.3.1.2.2** Synchronize Particles Synchronize particles is required due to the fact every module has a local list of particles to allow parallelization. These particles are updated, yet only one set of dynamics or stellar evolution will be run at a time which prevents any overlapping of data modification. To synchronize the particles you run all the channels created to synchronize all the data to the master list of particles. After the master list is up to date, you then synchronize using all the channels to the modules. This leaves all the modules' list of particles with the latest version of particle data.
- **5.3.1.2.3 Run Diagnostics** The function should loop through all the diagnostic scripts and if the diagnostics should use the shouldUpdate function with the current state of the experiment to check if it should run. If the diagnostic should run, it then runs the update method passing in the master list of particles.(Figure 4.12)

- **5.3.1.2.4** Run Logging The function will loop through all the logging scripts and run any real time logging scripts that need to be run. The real time logging scripts are denoted by name and run their logData method passing in the list of particles.(Figure 4.13)
- **5.3.1.2.5 Increment Time** The time should increment using the specified timestep (Figure 4.1). The stopping of the experiment is signaled when the stopping conditions are met. (Satisfies Requirement 2.4.3.1)

5.3.1.3 Post Simulation

- **5.3.1.3.1** Run Closing Diagnostics Scripts The function should loop through all the diagnostic scripts and if the diagnostics should use the shouldUpdate function with the current state of the experiment to check if it should run. If the diagnostic should run, it then runs the update method passing in the master list of particles.(Figure 4.12)
- **5.3.1.3.2** Run Closing Logging Scripts The function will loop through all the logging scripts and run any post simulation logging scripts that need to be run. The post simulation logging scripts are denoted by name and run their logData method passing in the list of particles.(Figure 4.13)
- **5.3.1.3.3 Stop Modules** The modules should be cleaned up after being run. This involves if running in parallel sending a kill command via MPI. We will run the modules' (AMUSE class) stop() command.

6 Networking

6.1 Overview

The networking layer is responsible for communication between the interface and the cluster status daemons running on the cluster nodes. The user interface and console can send messages through the network which will be distributed via IP multicast to any cluster nodes listening to a selected multicast group. Cluster nodes will be able to send replies to the sender containing information such as health stats and the number of experiments running on that node. The UI and command line tools will use IPC (pipes etc...) to communicate with local instances of the control program to send network packets to the cluster nodes.

6.2 Network Software Components

6.2.1 Control Instance

A control instance is a program that will run in the background on any machine wishing to control experiments on a cluster. The core scripts (command line tools)

and UI will both communicate with this instance to perform any network access.

6.2.2 Node Instance

Node instances are processes that run on each computing node in the cluster. The node instance tracks health statistics and running experiments and can send these pieces of information as direct (unicast) replies to queries received through the multicast group.

6.3 Object Model

6.3.1 Overview

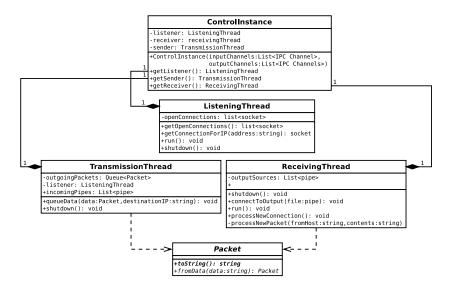


Figure 6.1: Network Object Model

6.3.2 Control Instance

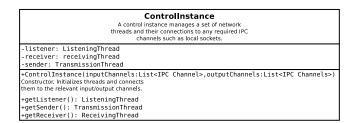


Figure 6.2: Control Instance Class Diagram

6.3.3 Node Instance

NodeInstance

A node instance runs once on each node in the cluster that needs to be managed. Each node instance listens for multicast packets and replies to requests for information.

-nodeName: string

The name of the node that will be returned if requested in a status query.

+NodeInstance(multicastGroupIP:string, nodeName:string)
Constructor called when the node instance is
started. It connects to the desired multicast
group and listens for requests.

Figure 6.3: Node Instance Class Diagram

6.3.4 TransmissionThread

The transmission thread (Figure 6.4) will process outgoing packets queued for transmission by external sources such as the UI.

TransmissionThread

The transmission thread listens to a set of IPC channels and gathers strings that need to be sent to other machines. These strings are then sent out over the network, establishing a connection if needed.

-outgoingPackets: Queue<Packet>

Thread-local queue of Packets that other threads/programs have queued to send.

-listener: ListeningThread

The TransmissionThread needs access to the listener in order to access established

+incomingPipes: List<pipe>

Programs that wish to send data can place the packet strings into one of these pipes. The transmission thread will collect the strings and send them to the destination.

+queueData(data:Packet,destinationIP:string): void
+shutdown(): void

Figure 6.4: Transmission Thread Class Diagram

6.3.5 ReceivingThread

The receiving thread (Figure 6.5) will read incoming packets from established connections with node instances.

ReceivingThread The receiving thread collects sockets from the listener and performs select() calls on them to gather incoming packets. These packets are distributed to any other programs on the system that need them, via IPC channels. Programs that need to be notified of incoming traffic can connect to a pre-defined pipe to receive data from the network. These pipes are stored in the receiver thread so it can send copies of each packet to the programs on the other side of the pipe. +shutdown(): void Cleanly shutdown the thread, closing any established connections. +connectToOutput(file:pipe): void Adds the specified file descriptor to a list of file descriptors that need data from the eceiving thread. +processNewConnection(): void When a new connection is established in the listener, it uses this method to inform the receiver that it needs to add this connection to the list of potential remote senders. -processNewPacket(fromHost:string,contents:string) Handles received data from a single host, called whenever the thread receives new data from a onnection.

Figure 6.5: Receiving Thread Class Diagram

6.3.6 ListeningThread

The listening thread (Figure 6.6) will listen for and accept new incoming connections from node instances.

ListeningThread The listener thread will run accept() in a loop and pass off new sockets to the receiver thread which will select/read the sockets to collect data. -openConnections: list<socket> +getOpenConnections(): list<socket> Returns a list of all connections accepted by this listener. +getConnectionForIP (address:string): socket Given an IP address, this finds the corresponding socket for that connection and returns it. +run(): void +shutdown(): void Cleanly shuts down the thread, closing any open connections.

Figure 6.6: Listening Thread Class Diagram

6.3.7 Packet Objects

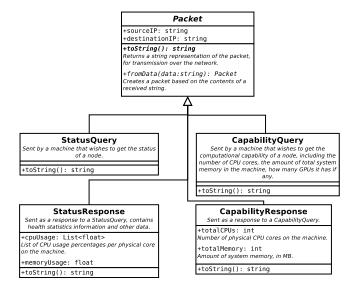


Figure 6.7: Packet Classes

7 Data

7.1 Experiment Specification

The following is a list of XML specifications for experiments. Experiments are saved and loaded via this format. Note that not all specified attributes will always be present because not all attributes are relevant to every experiment. See Figure 7.1 for an example XML experiment file. (Satisfies requirement 2.3.3, 2.3.5)

Figure 7.1: Example XML Experiment File

7.1.1 Experiment

Experiment tags are the first and last tags in an experiment specification because they encapsulate all experiment attributes.

Attribute	Description	
name	The name of the experiment	

7.1.2 Time

Time tags specify the time parameters of the experiment.

Attribute	Description
units	The time units for the experiment to use, defined by AMUSE
start	The start time of the experiment
step	The amount of time between simulation snapshots
end	The time to end the experiment

7.1.3 Module

Module tags define the modules used in the experiment. Each module has child param tags to define initial conditions. These initial conditions are defined by AMUSE.

Attribute	Description
name	The name of the module to use

7.1.4 Param

Param tags define the parameters for their parent model. Each defines a model's parameter value for use in the experiment. All parameters, default values, units, and descriptions are defined by AMUSE.

Attribute	Description	
name	The name of the parameter to initialize	
value	The value to give the parameter	

7.1.5 Particle

Particle tags define the particles used in the experiment. All references to units are those defined by AMUSE.

Attribute	Description
mass	The mass of the particle with units
radius	The radius of the particle with units
position	The position of the particle in [X,Y,Z] format with units
velocity	The velocity of the particle in [X,Y,Z] format with units
luminosity	The luminosity of the particle with units
temperature	The temperature of the particle with units
age	The age of the particle with units
stellarType	The stellar type of the particle, defined by AMUSE

7.1.6 Diagnostic

Diagnostic tags define the diagnostic scripts used in the experiment.

Attribute Description	
name The name of the diagnostic for reference pu	
file The filename of the diagnostic script	

7.1.7 Logger

Logger tags define the logging scripts used in the experiment.

Attribute	Description	
name	The name of the logger for reference purposes	
file	The filename of the logging script	

7.1.8 Stopping Condition

Stopping Condition tags define the stopping conditions used in the experiment. All stopping conditions are defined by AMUSE.(Satisfies Requirement 2.4.3, 2.4.3.2)

Attribute	Description	
type	The type of stopping condition to use	

7.2 Network Packets

The following is a list of packet data content specifications. Data is described in terms of types to ease implementation, however the information is sent over the network as a string to avoid any machine-specific byte ordering issues. Fields in the packet are separated by a "|" character. If this character is to appear as text inside a packet for any reason, it must be escaped as "\|"

7.2.1 Packet Header

The packet header precedes data in all packets.

PACKET_TYPE	LENGTH	SOURCE_IP	DEST_IP
int	long	string	string

7.2.2 Status Query Packet

The status request may contain flags requesting additional data from the node beyond the elements specified here. If the recipient understands the flags, they will fill the ADDITIONAL DATA field with the appropriate response.

HEADER	ADDITIONAL_	_REQUEST_	_FLAGS
-		string	

7.2.3 Status Response Packet

HEADER	CPU_USAGE	MEMORY_USAGE	SIMS_RUNNING	ADDITIONAL_DATA
-	float	float	string	string

7.2.4 Capability Query Packet

The additional data protocol is the same here as in the status request above.

HEADER	ADDITIONAL	_REQUEST_	FLAGS
-	string		

7.2.5 Capability Response Packet

HEADER	NUM_CPUS	MAX_MEMORY	NUM_GPUS	ADDITIONAL_DATA
-	int	long	int	string

8 Appendix

8.1 Requirements Traceability Table

Requirement	Section in SDD
2.2.1	4.3.1
2.2.2	4.3.1
2.2.3	4.3.1
2.2.4	4.3.1
2.2.5	4.3.1
2.3.1	4.2
2.3.2	4.2
2.3.3	4.1
2.3.4	5.3.1.1.1
2.3.5	7.1
2.3.6	4.3.2
2.4.1	5.2
2.4.2	5.3.1.2
2.4.3	5.3.1.2.5
2.4.4	5.3.1.2.5
2.5.1	5.3.1.1.4
2.5.2	5.3.1.1.4
2.6.1	4.4
2.6.2	4.4
2.6.3	4.4
2.6.4	4.4
2.6.5	4.4
2.6.6	4.4
2.7.1	4.5
2.7.2	4.5
2.7.3	4.5
2.8.1	3.3
2.8.2	3.3

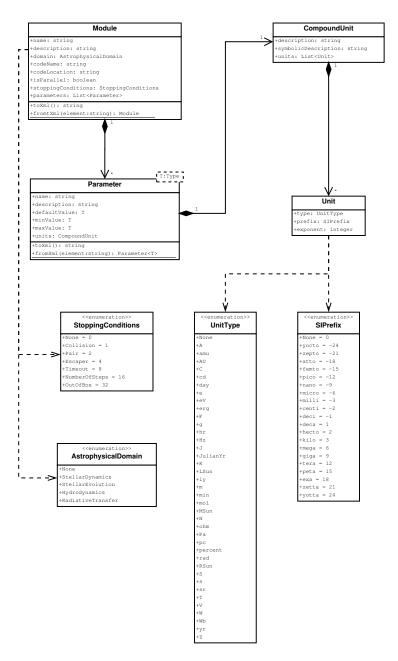


Figure 4.3: Relationships between Module and supporting classes