

# Network Ferret™

## Programmer Guide

### v12.4



# Contents

Preface .....	i
Network Ferret License Agreement.....	i
About This Guide.....	i
Who is This Guide For? .....	i
Getting Additional Help.....	i
Getting More Information .....	i
Getting Support.....	ii
1: Getting Started - Concepts .....	1
System .....	1
Agent.....	1
Master Agent vs. Subagent.....	2
Agent State.....	2
Config.....	2
Host.....	4
Processor .....	4
State Machines.....	4
Vendor Architecture.....	5
Logging .....	5
File Handling .....	5
Network Scanner.....	6
2: Basic Coding Examples.....	7
System .....	7
Agent.....	7
Subclass Agent.....	8
Agent State.....	8
Subclass Agent State .....	9
Config File Parameters.....	10
Hosts .....	10
Processors .....	10
Basic Processor.....	11
Synchronous Processor.....	12
SNMPTableProcessor .....	14
3: Types of Systems .....	16
Starting a multi-agent system .....	16
Things To Be Aware Of .....	16
Starting a single-agent system .....	17
Starting a “lazy” multi-agent system .....	17
Vendor Specifications.....	17
4: Pingging .....	18
PingAgent.....	18
Starting The PingAgent & Config Parameters.....	18
Synchronous Ping.....	19
Asynchronous Ping.....	20



Stopping The PingAgent .....	20
5: Credentials .....	21
Providing/Receiving Security Information .....	21
Possible Security Information .....	21
Credential Interface .....	21
Default Credential Interface .....	22
SNMP Credential Logic .....	23
6: Logging/Debugging .....	24
Logging .....	24
Debugging .....	24
Agent/Processor Debug .....	24
SNMP Debug .....	25
Java Memory Dumps .....	25
7: Protocols .....	26
SNMP and Buffers .....	26
High Volume .....	26
GETBULK .....	27
NetConf/REST .....	28
8: Utilities .....	29
CacheMember .....	29
Converter .....	29
DNSLookup .....	29
File .....	29
IFFilter .....	29
IPAddress .....	29
Queue .....	29
Reflection .....	30
9: Getting Started With Network Ferret .....	31
Before Writing Any Code .....	31
Network Ferret Architecture .....	31
Wire Model .....	33
Customization Options .....	33
Customizing basic discovery .....	33
Customizing advanced discovery .....	34
Customizing vendor-specific information .....	34
10: Embedding the Discovery Engine .....	35
Environmental issues .....	35
Platforms and JREs .....	35
Java Native Image .....	35
Third Party Jars .....	36
Working with the configuration files .....	36
Running discovery programmatically .....	36
Running from within a Java program .....	36
Running as a standalone process .....	37
Handling discovery output .....	37
Registering with Network Ferret .....	37



Receiving progress and debug messages .....	38
Receiving error messages .....	38
Address parameters in messages .....	39
Receiving domain information.....	39
Error handling .....	42
Controlling a running discovery .....	43
Pausing discovery.....	43
Resuming discovery .....	43
Stopping discovery .....	43
Running Concurrent Discoveries.....	44
Querying Discovery For Progress .....	44
Providing Security Information.....	45
Debugging while using Network Ferret.....	45
11: Extending Basic Discovery .....	46
12: The Wire Model .....	47
Design Considerations .....	47
Combining Discoveries.....	48
Serialization .....	48
Size of the Wire Model.....	49
Knitting.....	49
Performing a Knit.....	49
What Is and Is Not Analyzed on a Knit .....	50
Accessing The Wire Model .....	50
13: Creating An Advanced Discovery.....	51
Advanced Discovery Overview.....	52
14: Localization.....	53
Setting the Locale.....	53



# Preface

This preface contains background information that you should know before using this Guide.

## Network Ferret License Agreement

Read The Terms And Conditions Of The License Agreement Found In The Network Ferret User Guide.

## About This Guide

The chapters of this guide contain explanations, reference information, and examples of how create code independent of Network Ferret (using NMSCore), how to embed Network Ferret and how to extend Network Ferret functionality by writing Java code.

## Who is This Guide For?

This Guide is intended for programmers who want to embed Network Ferret in their application or write Java code to extend the functionality of Network Ferret.

This Guide assumes that readers already understand:

- How to program applications in Java.
- The basics of SNMP, Network Management.
- The basics of using operating systems such as Microsoft Windows and Unix.

This Guide assumes readers already have:

- Access to a copy of the Network Ferret installation media.
- Become familiar with running Network Ferret and analyzing the output it produces.

## Getting Additional Help

The following sources of additional assistance and information are available to all Network Ferret users.

## Getting More Information

You can find additional information about Network Ferret in the following related publications:

- *Network Ferret User Guide*: Intended for users who want to understand how to run Network Ferret.



- *Network Ferret Domain Model*: Intended for all users who want to understand the data that Network Ferret generates. This includes general models as well as models generated by vendor-specific device handling.

## Getting Support

When contacting support, be prepared with the necessary information that will make handling your problem easier. You should have:

- Log.txt (or your own log) – this may be called something else by your application. This file can get rather large. If shipping the file to support, it is best to zip it up. If the problem is regarding a specific device, support may have you sort the log and pull out the information for that device rather than sending the entire log.
- Errors.txt – this may be called something else by your application.
- Exceptions.txt – this may be called something else by your application.
- In some cases support may ask you to zip up the config, log and report directories.

The full name of the logs will be prefixed by the Agent's name. For Network Ferret this will be "discovery".



# 1: Getting Started - Concepts

The first part of this Guide will be about the general programming architecture provided by NMS Core. For 20+ years we have wanted to generalize the architecture so applications could be created independent of auto discovery (Network Ferret). We were leery to do so for fear of breaking something that was working and for always thinking of some reason a new design would not work as well.

Here we will introduce some basic concepts.

**If you simply want to embed Network Ferret without doing any of your own extensions or independent agents, start with the chapter: [Getting Started With Network Ferret](#).**

## System

System maintains “global variables and Agents” to be shared by all Agents running in the VM. Examples of global variables/agents/services are the one PingAgent, the one SNMPCredentialAgent and the one CredentialInterface. System can be subclassed to add custom global variables.

Other services AtiSystem can provide are a global MIB Dump Agent and a global DNS lookup service.

AtiSystem will determine the platform architecture and the default gateway for the machine.

AtiSystem parses the OUI file from the config directory and provides a few methods to translate MAC prefixes into a vendor string from the file.

AtiSystem provides a simple Agent registry. A dictionary where key is some string and value is the Agent instance. This allows Agents to find each other and communicate.

## Agent

Network Ferret is an Agent. It is a complex collection of logic and threads of execution. Other Agents might be something like an agent



that collects time series data for performance analysis or an agent that maintains configurations of network devices.

There can also be subagents which are an agent that has a master agent. For example, the L2 and L3 advanced discoveries in Network Ferret are subagents of the main discovery agent.

## Master Agent vs. Subagent

Any Agent can have a master agent. The agent inherits the AgentState from the master and will be shutdown when the master shuts down. An example of this are the System agents (ping, credential and mibDump). These agents can be started independently or they can be part of a mater agent.

Network Ferret, for example, will check to see if a System ping agent exists and, if not, create one that is a slave to Discovery. When discovery finishes, the ping agent will be stopped.

A subagent is a bit similar. It also inherits the AgentState from the master. However, a subagent is more tightly tied to the master in terms of programming logic. The master feeds Hosts into the sub agent's queue. The master is monitoring the subagent for status/completeness. In Network Ferret, the L2 and L3 advanced discoveries are subagents.

## Agent State

This concept existed in previous versions of Network Ferret.

The Agent State is a subclass of Java ThreadGroup. This allows Threads created deep in the application logic to still have access to "global variables". Agent State also takes away the need for global static variables defined in classes which can get messy and cause bugs if multiple instances of an Agent are running or different agents are sharing the same classes.

In theory, all subordinate threads should be stopped when the main thread exits, but this is not always the case. Threads waiting on something generally need to be bounced off the wait in order to exit.

## Config

Config files exist in previous versions of Network Ferret, but the concept has been expanded in V12 to provide more flexibility.





The Config is essentially a dictionary of key/value pairs parsed from the config files for each Agent. The Agent State holds on to the Config thus making it accessible to all threads in the Agent.

### **Where do config files come from?**

AtiSystem has fields for root and config directories. These are used as a last resort. But these are useful if the configs for all Agents are in the same location. That depends on how you design your system. If one group has control over all Agents, then it might be best to have all configs in one location. If Agents are controlled by multiple programming groups or 3<sup>rd</sup> parties, it may be best to segregate config/log directories.

When an Agent is started, root and config directories can be passed in. If these are NULL, then the AtiSystem directories are used. If these are NULL, nothing happens. No files are read.

The first file read is defaults.cfg found in the AtiSystem config directory.

The second file read is defaults.cfg found in the Agent config directory

The third file read (optionally) is the file returned from the method agentConfigFileName(). This can be NULL. Consider this file to be defaults for the specific Agent. In the distribution you can find config files for ping, credentials, discovery, etc.

The final file read is the config file passed in when the agent is started. These are run-time parameters that will override any values in the other files. This will also include parameters that simply do not exist in the other files such as which IP addresses to work with in this run.

### **Why two defaults.cfg??**

For flexibility. Suppose a 3<sup>rd</sup> party was distributing a collection of agents. This 3<sup>rd</sup> party may want its own defaults.cfg for its collection of Agents.

All of the config files can be bypassed for an Agent by passing in a Properties object instead. For this reason, any Agent should have hardcoded defaults that match the config file defaults so that the code starting the Agent only must specify parameters it wants to change and not ALL parameters.



# Host

A Host represents an IP address being worked with. It maintains credentials for various protocols, statistics, etc. This has existed in previous versions of Network Ferret.

# Processor

A Processor is a thread of execution. **This is a big change from previous versions of Network Ferret where the thread of execution and the domain logic were bound together.** No more.

**If you have written your own Network Ferret extensions, these will need to be modified.**

For example, the SNMPTableProcessor has been split into the SNMPTableProcessor (thread of execution) and the SNMPTableWalker (table walking logic). An SNMPTableWalker can be used with or without an SNMPTableProcessor.

This allows two competing styles of architecture. Network Ferret is built to scale massively while minimizing OS resources. Network Ferret uses Processors that allow many Hosts (IP addresses) to share the same thread of execution and SNMP socket.

With NMSCore, an architecture can be constructed which is of the style of a single thread of execution and socket per IP address to be worked with.

# State Machines

Network Ferret processors usually had state machine logic built into them. It was just the nature of auto discovery. For example, discovering MPLS requires querying multiple tables with a different path through the tables depending on what is found and not found. Then all the data has to be analyzed after querying the tables is completed.

This type of complex logic has been pulled out and generalized into State Machines with States and SNMP State Machines with SNMP States.

State machines are logic and NOT a thread of execution.



# Vendor Architecture

Even though many standards are defined, vendors tend to either implement the standards differently or implement things in private MIBs. Protocols like NetConf and REST are certainly vendor-specific.

Network Ferret provides a file-based architecture where vendor-specific key/value pairs as well as Java classes can be defined. The files are related to a given Host based on the SNMP sysOID although this is not strictly required.

They are called VSP (Vendor Specification) files and can be found in the config/vendor directory of the distribution. The root VSP is called, unsurprisingly, root.vsp. There is also an SNMPRoot.vsp.

The CredentialAgent not only determines the valid credential for a Host (address) but also determines the proper VSP for the Host. Once the VSP is set, it is possible to load code defined specifically for the vendor/device type, look at flags defined in the VSP, etc. The Credential Agent will also test if a host supports **GETBULK** or not.

Note that **GETBULK** can be disabled in the VSP files for a given device. This is because, even though a device may support **GETBULK**, some of them behave very strangely and **GETBULK** must be disabled.

AtiSystem maintains a global VSP database but an Agent is permitted to have its own.

## Logging

Network Ferret provides both logging and reporting infrastructures. Agents can each write to their own log or one master agent can start all of the other agents and they could share the same log.

See defaults.cfg for a few parameters that control how/where logs are named/created.

## File Handling

AtiFile in the utils package provides some methods for dealing with Java serialized files and other files. It assumes the “report” directory for files with no path specification.



# Network Scanner

V12.3 contains a new package called NetworkScanner. These are abstract classes which help to create a long-running agent that is scanning some data on a periodic basic. See the Java doc.

An example of this is an Agent that is scanning switch MAC tables looking for MACs coming and going.

Network Ferret does not use this package since discovery is a point-in-time system. It is not long-running.



## 2: Basic Coding Examples

This chapter gives some basic coding examples. Nothing too complex. The goal is to give a flavor of what the coding might be like. Note that there is Java source code for the examples in the distribution.

### System

AtiSystem maintains global variables (see previous chapter) to be shared among Agents within the VM. There is no instance. All methods are static. See the various sections that have objects here such as Ping and Credentials.

### Agent

The Agent provides the main thread of execution. Even if an application is a single Agent started from main(), the Agent will have its own thread of execution.

Below is an example of running the Discovery agent within an embedding application. In this example, the embedding application does nothing more than run discovery.

The execute() method simply starts the main Discovery Agent thread and then waits for it to complete. A more complex embedding application would start the Discovery Agent asynchronously and then do something else.

```
public static void main (String[] args)
{
    AtiIPDEmbeddingExample e = new AtiIPDEmbeddingExample();
    e.execute();
}

public void execute()
{
    AtiIPDDiscovery d;                // Subclass of AtiAgent

    String rootDir = "c:\\amt";        // Set depending on where you installed
    String configDir = "config";
    String configFile = "home.cfg";

    // Create the discovery instance
    d = new AtiIPDDiscovery();
```



```

        // Run discovery. This will not return until discovery is complete.
        // An alternative is to call executeAsynch(); which will return
immediately.
        d.execute(rootDir, configDir, configFile);
        System.out.println("DONE EMBEDDING EXAMPLE");
    }

```

## Subclass Agent

AtiAgent is abstract and MUST be subclassed. See Javadoc for details.

Here are the main items you need to worry about in your subclass. Other things are discussed in other sections such as Agent State and Host.

There are the `name()` and `setName(String name)` methods. You can choose to override the first or call the second.

The `run()` method is final. You will not override it. Run does the following:

Parse the config files, open the log files.  
Call `void logCopyright()`

Call `boolean preRun()` which you may override. If you return false, `executelt()` will NOT be called.

Call `executelt()`. **This is what you override.** The default implementation does nothing. This method should not return until the Agent logic is finished.

Call `void postRun()` which you may override. The default for `postRun()` and `preRun()` is to do nothing.

Closes the logs.

## Agent State

AgentState is a subclass of Java ThreadGroup. The Agent thread and all threads it creates, or its children create, are part of the ThreadGroup. If your Agent provides an external API to access data in the Agent, most likely, the threads making those API calls WILL NOT BE PART OF THE THREAD GROUP. For example, Network Ferret had to keep some variables as fields in the Discovery object (Agent) rather than the AgentState so they would be accessible to external threads making API calls.



The Agent initialization creates the AgentState, assigns itself to the **agent** field in AgentState, creates the main thread, attaches it to the AgentState, and starts the thread which calls run().

Any code can access the AgentState via **AgentState.getState()**.

See the Javadoc for the details of what AgentState has available. Most things shouldn't be touched because they are used by the infrastructure. But there are a few useful things:

- The Agent instance (**AgentState.getState().agent**)
- The Config instance (**AgentState.getState().config()**)
- Get an SNMP PDU (**AgentState.getState().getPdu()**)
- Get an SNMP Poller (**AgentState.getState().getSnmpPoll()**)
- String cache
- File lock object

The reason for the PDU and Poller is because some clients wanted to subclass these and add functionality, but they had no ability to modify the infrastructure to use them.

The File lock object is something special that Network Ferret needed. Some NF classes load/parse files during class initialization. If multiple instances of NF were running within one Java VM, this could cause a problem. Since the Class is initializing, there are no variables available to use as a lock object. The Agent State provides this object.

You can also add/retrieve private objects to the AgentState using the **privateSet()** and **privateGet()** methods. Or you can subclass AgentState and add new fields.

## Subclass Agent State

Agent State can be subclassed. AtiAgent calls its own method below which your AtiAgent subclass would override to provide a subclass of AtiAgentState

```
protected AtiAgentState initializeAgentState(String threadGroupName)
{
    return new AtiAgentState(threadGroupName);
}
```

Subclass AtiAgentState because you want to provide other “global variables” to your application logic. Also, if there are high volume access parameters in the AtiConfig, you might want to transfer them to fields in the State to minimize the number of hash table lookups.



# Config File Parameters

Config file parameters (Key/value pairs) are parsed and kept in an AtiConfig object. See the Javadoc for access methods.

The Config object is accessible via:

`AtiConfig.config()` or  
`AtiAgentState.getState().config()`

If two different threads access the Config object and you get two different Config objects, it means one of your threads is NOT in the Agent ThreadGroup.

Processors have some special handling of config file parameters. See that section.

## Hosts

AtiHost represents an IP Address that is being interrogated. A Host maintains credentials for the supported protocols (SNMP, NetConf, REST), various statistical counters (like send/rcv bytes) and a few other things.

There is an entire credential mechanism for storing, retrieving and discovering credentials. This is discussed later.

The simple, hardcoded, way to create a host is:

```
AtiHost host = new AtiHost("10.10.10.1");
```

Remember to go through the credential verification process (CredentialAgent) as this process not only determines a valid credential but also determines the VSP file for this Host.

A Host does have a discard flag which gives a reason for the discard which is usually a timeout during a query. See the Javadoc.

The SNMP credential determination process will also retrieve the sysOID and set the vendor profile (AtiVendorSpecification) for the Host.

## Processors

Processors provide a thread of execution and will be the heart of any complex Agent.

A number of possible architectures are possible with Processors.





Network Ferret uses processors that handle a large number of Hosts concurrently in order to scale without using up local OS resources (threads and sockets).

A smaller scale Agent could use a processor as a thread of execution for a single Host. So, the Processor would take the Host in and all logic for the Host would be executed in that Processor thread and, when finished, the processor would exit. Multiple Processors could be started to handle multiple Hosts concurrently.

A third possibility is to use no Processors and do all of the work out of the Agent's thread of execution.

There are other combinations of processors that are possible.

Processors can be chained together and they have logic for querying up and down the chain to determine the "progress" of the chain. Network Ferret uses this logic to provide feedback to the embedding application as to the progress of auto discovery.

## Basic Processor

The most basic Processor is called `AtiProcessor`. All other Processors are subclassed off of this. It is an abstract class.

Every Processor will have an `AtiQueue` and a `Thread`. Some external logic will add `AtiHosts` to the queue which will be pulled off and have the following methods called which concrete subclasses should override.

```
protected void runIt(AtiHost host)
{
    if (this.processHost(host)) // give subclass a chance to reject
        this.beginHost(host); // subs MUST call finishedHost()
    else
        this.finishedHost(host);
}
```

Use this for the most basic of logic or a basic architecture where sequential processing is acceptable regardless of how long it may take.

It is possible to use sequential processing AND create a processor per Host being processed to create a multitasking effect.



## Synchronous Processor

The Synchronous Processor is a multitasking processor that will create a thread for each host being processed. It is essentially the same paradigm as using a basic `AtiProcessor` per Host to be processed.

The difference is that the Synchronous Processor works nicely with a multitasking architecture that uses other multitasking processors such as the `AtiSNMPTableProcessor`. In this case the managing Agent can easily construct and manage a chain of processors. This is how Network Ferret handles things like NetConf queries which are synchronous.

Use the `SynchronousProcessor` when using any protocol that is going to block the thread and wait; such as TCP.

`AtiSynchronousProcessor` manages a list of `AtiSynchronousProbes` (the thread of execution for each Host – it actually extends `Thread`).

### Using `SynchronousProcessor` directly

It is not necessary to subclass `SynchronousProcessor`. In this case, use this constructor:

```
/**
 * Constructor for when you are NOT subclassing this class.
 *
 * @param inAgent
 * @param probeClass The subclass of AtiSynchronousProbe we will instantiate
 * @param maxHosts The maximum number of concurrent hosts to be processed
 */
public AtiSynchronousProcessor(AtiAgent inAgent, Class<?>probeClass, int maxHosts)

    new AtiSynchronousProcessor(myAgent, myProbeClass.class, 0);
```

Pass in 0 to take the default number of maximum hosts which is 10. A probe will be created for all Hosts (there is no `processHost()` to call).

### Using a subclass of `SynchronousProcessor`

The method `processHost()` should be overridden just like for `AtiProcessor`.

This method may be overridden. This defines the maximum number of concurrent Hosts to be processed.

```
protected int maxHosts()
```



This method **must** be overridden:

```
AtiSynchronousProbe getProbe(ThreadGroup tg, String name);
```

A typical implementation is like this from the  
AtiNetConfCredentialProcessor class.

```
public AtiNetConfProbe getProbe(ThreadGroup tg, String name)
{
    return new AtiNetConfCredentialProbe(tg, name);
}
```

Finally, these two methods can be overridden. These are timeouts for the probes.

```
public long connectTimeout() - default is 10,000 ms
public long workTimeout() - default is 10,000 ms
```

## SynchronousProbe

AtiSynchronousProbe extends Thread. This is a generalization of the run() method for the Probe:

```
public final void run()
{
    setThreadName(); // Give sub a chance to override
    if (connect())
        doWork();
    cleanup();
    AtiDebug.msg(host.address(), name(), "Done run()");
}
```

**initialize()** is called before the thread is started.

Subclasses MUST implement **connect()** and **doWork()**. **Cleanup()** does not have to be overridden unless the subclass has something to cleanup.

There are two flags that subclasses need to set/watch. The **shouldExit** flag will be set when **shutdown()** is called which is usually called by the **kill()** method. The **connect()** and **doWork()** methods should monitor this flag and short-circuit execution when it is set to true.

The **isConnected** flag is set to true by the **run()** method when **connect()** returns true. It is up to the subclass to change this to false, if necessary, during execution.



## SynchronousProbe – Timers

SynchronousProbe puts timers on both the `connect()` and `doWork()` calls. It calls the methods `connectTimeout()` and `workTimeout()`. The default behavior is to call the SynchronousProcessor `connectTimeout()` and `workTimeout()` methods.

Either the SynchronousProbe subclass or the SynchronousProcessor subclass can override these methods. The default timeout for both is 10 seconds. A return value of  $\leq 0$  means no timer is set and it is up to your code to deal with it.

Override the processor-level methods if timeouts are the same for all probes. Override the probe-level methods if, for some reason, timeouts can be different per probe.

## SNMPTableProcessor

This processor provides a single thread of execution and a single SNMP socket to be shared by multiple instances of AtiSNMPTableWalker (logic). Network Ferret uses these so it can scale massively.

There are three flavors of SNMPTableProcessor. A processor that runs state machines, a processor that runs the same class of SNMPTableWalker and a processor that runs different classes of Walkers/StateMachines based on a tag found in the vendor VSP files. Network Ferret uses them all. Here is a code snippet that starts two types of SNMPTableProcessor plus other types of AtiProcessor. Exception handling has been removed for ease of reading.

```
// Create the processor instance

// Class is a Walker class
if (processorClass.getSuperclass() == AtiIPDSNMPTableWalker.class ||
    processorClass.getSuperclass() == AtiIPDSNMPTableWalkerCS.class)
{
    p = new AtiSNMPTableProcessor(this, processorClass);
}
else if (className.endsWith("StateMachine")) {
    // The class is the class that creates the state machines. We create
    // one instance and pass it in to an SnmpTableProcessor.
    Object smf=null;
    smf = processorClass.newInstance();
    // Assume class's StateMachine factory method is getStateMachine()
    p = new AtiSNMPTableProcessor(this, smf, "getStateMachine");
}
```



```
else {          // Some kind of AtiProcessor that is not an SNMPTableProcessor
    Constructor<?> cArray[] = processorClass.getConstructors();
    Constructor<?> c = cArray[0]; // there will only be 1
    Object args[] = new Object[1];
    args[0] = this;
    p = (AtiProcessor) c.newInstance(args);
}
```



# 3: Types of Systems

See the FullAgent example source code shipped with the product. This gives lots of comments and various options for starting the system.

## Multi-agent system

Suppose you are running in some execution environment and you have multiple Agents doing various things at various times. In this case it would be best to have global instances of the Ping, Credential and MIBDump Agents as well as a global instance of the Vendor Specification database (the CredentialAgent loads this).

Make sure the System starts these Agents independent of another agent. Otherwise, that agent will own them.

## Things To Be Aware Of

**Log Files** – Because all of these agents are in their own ThreadGroups, they MUST have different log file names. As long as no names are specified in the config files, all is OK because each agent will default to using its own log file names.

**Threads Of Execution** – Because all of these agents are in their own ThreadGroup, they all have their own AgentState. The Ping, Credential and MIBDump agents all have the ability to do a callback to provide a result. KEEP IN MIND, that your callback method is being executed by that Agent's thread and WILL NOT HAVE ACCESS TO YOUR THREADGROUP. Therefore, the callback method cannot do anything that involves accessing the AgentState. This includes log messages. For example, if you receive a callback from the Ping agent and log the ping result in the callback, it will end up in the Ping Agent's log.

Network Ferret was written assuming that these Agents are in their own ThreadGroup. For example, Network Ferret does a lot of work with a ping result so when it gets the callback from the PingAgent, all the callback method does is queue the response so Network Ferret's thread can do the processing.



## Single-agent system

A single Agent system is much easier to code than a multi-agent system. In a single Agent system, all Agents share the same AgentState. There is no issue with log files or threads of execution/AgentState as with the multi-Agent model.

Discovery, for example, will start all of the System agents if they are not already running. All messages from all agents go to the discovery log and not to individual agent logs.

## “Lazy” multi-agent system

All Agents could be coded to start their own private copies of Ping, Credential, etc. Your Agent goes not care if it is in a multi-agent system but resources are not used as best as they could be. Credentials may be checked multiple times. Multiple raw ICMP sockets may be opened.

**There currently is a potential problem with doing this.** An instance of CredentialAgent loads the VendorSpecification database globally. See the next section.

## Vendor Specifications

Vendor Specifications are essentially constants so loading them globally is OK regardless of the type of system being implemented.

The Credential Agent will load the global vendor database.

Vendor Specifications can be local to the Agent but the only reason to do this is if there are entirely different sets of VSP files for some Agent. **LIMITATION:** An Agent cannot have a private CredentialAgent AND a set of VSP different from the global set.



## 4: Pinging

NMSScore provides an ICMP mechanism. Java provides a mechanism using `InetAddress.isReachable()` but this has several problems. First, there is no guarantee that the VM you are using will actually do a ping. Second, doing a ping requires a raw socket which requires admin privileges on some OSes. This means the entire VM must run with privileges which is risky.

NMSScore uses native Java code to use real ICMP packets and provides detailed results such as Host Unreachable, Network Unreachable, Administratively Prohibited, etc. For Unix systems, NF provides a small external executable that does the pinging. Only this executable requires privileges.

NMSScore's ICMP logic also deals with slowing down pinging if Source Quench returns come from routers out in the network.

NMSScore's ping mechanism also provides for the option to use a TCP port either after regular pings have failed or in place of regular pings.

### PingAgent

There probably should only be a single PingAgent in a VM but it is not required. AtiSystem holds on to an instance of PingAgent. Access it using `AtiSystem.pingAgent()`;

### Starting The PingAgent & Config Parameters

AtiSystem holds on to an instance of AtiPingAgent. How the instance gets there and what config parameters it uses depends on who starts it and how it is started.

See `ping.cfg` for parameters that the PingAgent uses.

#### **Independent PingAgent with default parameters and stdout logging**

Simply asking for the instance, `pingAgent()`, will create one if it does not exist.





```

public static void main (String[] args)
{
    // Start outside of any Agent thread group.
    AtiSystem.pingAgent();
    // rest of main code that starts an Agent(s)
}

```

### Independent PingAgent with its own configuration

Ask for the agent, the first time, and include the standard config directory/file parameters for an Agent.

```

public static void main (String[] args)
{
    // Start outside of any Agent thread group.
    AtiSystem.pingAgent("c:\\amt_12", "config", "ping.cfg");
    // rest of main code that starts an Agent(s)
}

```

### PingAgent started by another Agent – inherit AgentState & Config

If you know your Agent will be the only one creating the PingAgent, then the exists() check is not necessary. But if running in an unknown system, and some other Agent may have created the PingAgent, the exists() check should be done.

The PingAgent will write to the logs of whatever Agent starts it.

```

if (!AtiSystem.pingAgentExist()) {
    AtiIPDDebug.msg(name, "Starting Ping Agent...");
    AtiSystem.pingAgentSet(AtiPingAgent.start(this));
}

```

## Synchronous Ping

```

AtiIPAddress ip = new AtiIPAddress("192.168.1.254");
    // Synchronous ping
AtiPingRequest req = AtiSystem.pingAgent().ping(ip, null);
AtiPingResult res = req.result;
AtiIPDDebug.msg(ip.toString(), "TestPing", "Ping result code: " +
    res.getCode());

```



## Asynchronous Ping

```
AtiIPAddress ip = new AtiIPAddress("192.168.1.254");
AtiSystem.pingAgent().ping(ip, this);    // Asynchronous ping
// Code continues to execute here

// Callback method (implements AtiPingResultCallback interface)
public void pingReponse(AtiPingRequest request, AtiPingResult response)
{
    AtiIPDDebug.msg(new AtiIPAddress(response.getResponseHost()).toString(),
        "TestAgent", "Callback - Ping result code: " +
        response.getCode() + " id: " + response.getId());
}
```

Note that the PingAgent's thread is executing the callback method so any log messages will end up in the PingAgent's log which may be different from the caller's log.

An AtiQueue instance can also be provided in place of a callback method. In this case the ping result will be placed directly in the queue.

## Stopping The PingAgent

Any AtiAgent can be directly stopped with `stopNow()`. However, since AtiSystem is probably holding on to this instance, it is better to call:

```
AtiSystem.pingAgentStopIfIamMaster(AtiAgent master, boolean stopIfNoMaster)
```

If the PingAgent has a master agent and it matches the Agent passed in, the PingAgent will be stopped. Pass in TRUE for stopIfNoMaster to stop a PingAgent that was started with no master.

Passing in a master and TRUE will stop the PingAgent unless it was started by some other Agent.



# 5: Credentials

## Providing/Receiving Security Information

Credential handling (SNMP, VMWare, NetConf and REST) is part of the core infrastructure since multiple agents would want to share a valid credential rather than rediscovering the same valid credential.

The starting of the Credential Agent is discussed elsewhere. This chapter will only discuss the interface.

Note that the credential agent can be used by many agents. However, some code should be responsible for maintaining the credential interface which provides the Credential Agent with KNOWN credentials and stores new credentials that the Credential Agent discovers.

## Possible Security Information

It is documented in credentialAgent.cfg how to provide Network Ferret with a list of all **possible** security credentials for a given protocol.

When the Credential Agent needs to determine the credentials for a given IP address/protocol, it will first ask the credential interface for known credentials (see below) and then try the possible credentials.

This makes things easy on the administrator who does not need to know the specific security information for each IP address. The negative effect of this mechanism is that discovery runs a bit slower while the Credential Agent takes time to figure out which credential to use. Also, a large number of errors can be generated to existing management systems as various credentials are tried for each IP address.

With version 12.4, a method was added to CredentialInterface to allow the interface to provide possible SNMP credentials instead of using config files.

## Credential Interface

The Credential Interface is responsible for providing **known** credentials to the Credential Agent AND for storing new credentials that the Credential Agent discovers.



Known credentials will be tried first before any possible credentials (see above).

Some code needs to implement  
[com.logikos.core.credentials.AtiCredentialInterface](#).

```
public class myClass extends Object implements AtiCredentialInterface
```

The methods are as follows (See JavaDoc for details):

```
public boolean initialize();
public void addSNMPCredential(String address, AtiSNMPCredential snmpCredential);
public AtiSNMPCredential getSNMPCredential(String address, int version);
public void getSNMPPossibleCredentials(see Javadoc for parameters);
public void addNetConfCredential(String address, AtiNetConfCredential netconfCredential);
public AtiNetConfCredential getNetConfCredential(String address, int version);
public void addRestCredential(String address, AtiRestCredential restCredential);
public AtiRestCredential getRestCredential(String address, int version);
public void addVMWareCredential(String address, AtiVMWareCredential vmwareCredential);
public AtiVMWareCredential getVMWareCredential(String address, int version);
public void save();
```

The **add...()** methods are called when a credential is successful. For SNMP, the method could be called multiple times in a given discovery for a given address; once for each version of SNMP supported by the address. The Credential Agent calls this method even if the credential was initially obtained from the interface.

Use a version of 110 or 111 for NetConf credentials.

Use a version of 99 for VMWare credentials.

Use a version of 0 for REST credentials.

The **save()** method is called when the Credential Agent exits.

To register the interface you need to do the following:

```
AtiSystem.credentialInterfaceSet(yourInstanceOfTheInterface);
```

## Default Credential Interface

AtiSystem will use a default credential interface if none is set and someone tries to save a credential. The default will create a file in the config directory called cdb.dat. This file is protected by AES256 encryption and can only be read on the machine on which it was created. We have no way of debugging problems with this file.



## SNMP Credential Logic

For SNMP, the Credential Agent will follow the following logic:

- 1 – Ask the Credential Interface for a “preferred credential”. This could be any version. If a credential is returned, it will be tried.
- 2 – Ask the Credential Interface for a known credential for version 3.
- 3 – Try the possible credentials for v3.
- 4 – Repeat 2&3 for versions 2 and 1.

Only versions that are enabled via a config parameter will be tried. There are other config parameters which also control exactly how the credentials are queried.

Why a preferred credential? This is more efficient. Suppose all versions are enabled and IP address X has a known v2 credential. The Credential Interface could return this rather than the Credential Agent trying the possible v3 credentials before asking for the v2.



# 6: Logging/Debugging

## Logging

The `com.logikos.core.message` package provides a simple logging mechanism. Errors, Warnings and Debug messages. We should have called Debug messages Info messages, but there is too much code to change.

See the Network Ferret chapter below on [Handling Discovery Output](#) for more detail about the logging message callbacks.

Also, see the Network Ferret chapter on [Localization](#).

## Debugging

There are times when much more detail is required. Network Ferret v11 and earlier used to have a global “debug” flag but it created gigabyte logs because every part of the system was dumping detailed messages to the log. With v12 we have tried to make things a little more granular.

### Agent/Processor Debug

AtiAgent has a public variable called “debug”. Yes, it should be in the AtiAgentState but there was so much legacy code directly referencing it, that we decided to keep it. There are also methods to reference the flag. Code can also use `AtiAgentState.getState().agent().debug()`; You can see why there is much code directly referencing the variable. There are many, many debug checks in Network Ferret and we were concerned about making so many needless method calls.

There is an undocumented config parameter called `DEBUGIT`, which we are documenting here 😊. The AtiAgent class checks for it after parsing the agent config file and before `preRun()` is called. You can set this in `defaults.cfg` to turn on debugging for everything.

`DEBUGIT` can only be `TRUE`. In other words, if you manually set an agent debug flag at creation time, a false `DEBUGIT` value will NOT override this.

An AtiProcessor can have a local debug flag. If set, it will override the agent debug flag. If not set, the `debug()` method in AtiProcessor will return the agent debug flag.



## SNMP Debug

SNMP is a huge amount of Network Ferret work so it has its own debug flag. We did not want to clog the log with SNMP messages if you only want to debug your agent/processor logic. SNMP debug messages will go to stdOut.

The AtiAgentState maintains a debugSNMP flag. There is no config parameter to set this. Set this via code or your own agent's config parameter.

AtiSNMPTableProcessor, AtiSNMPTableWalker and AtiSNMPPoll all look for this at creation/start. Each of these classes has a debug flag which can be manually set. This allows you to be very specific about what code puts out debugging messages.

## Java Memory Dumps

AtiSystem has a public variable called "debugMemory". Some piece of code must set this to true. Various places in Network Ferret call the method AtiSystem.debugMemoryCheck("tag"). If debugMemory is false, nothing happens. If debugMemory is true, the calling thread will sleep for AtiSystem.debugMemoryPause milliseconds. The default is 20 seconds (20,000). This gives a programmer a chance to do a memory dump of the system.

For Network Ferret, we would generally do this after discovery was complete so we could see if we were hanging on to any objects we thought should be garbage collected.



# 7: Protocols

NMSScore has packages for SNMP, REST and NetConf. There is Javadoc for all of this.

The examples source code has a “protocols” package which shows how to use the various protocols. While NMSScore provides a great deal of support for high-level SNMP logic, you are certainly able to use the low level SNMP and “roll your own”.

Note that the **vijava jar** file is included in the distribution. Network Ferret uses this to discover VMWare systems. The CredentialAgent handles VMWare credential discovery. But there is no other infrastructure. You are free to use the jar API as you wish.

## SNMP and Buffers

When working with SNMP, there are some buffer issues that need to be watched.

### High Volume

If dealing with high volume, you must be concerned about overrunning the OS socket receive buffer. If this happens, packets will be silently dropped and they will appear as SNMP timeouts. This can be rectified by utilizing buffered SNMP. High volume will only occur when using an AtiSNMPTableProcessor which is handling many hosts at the same time.

Prior to v12, Network Ferret permitted the resizing of OS socket buffers but this was cumbersome. Setting buffered SNMP creates a separate read thread. All it does is read the OS socket and store PDUs for future reading.

AtiAgentState has a variable called useBufferedSNMP. This must be set by your code.

Since an individual SNMPTableWalker cannot overrun the buffer, by default, an SNMPTableWalker that is NOT part of an





SNMPTableProcessor will create a non-buffered SNMPPoll regardless of the value set in the AgentState.

## GETBULK

If using GETBULK, there is the issue of the Java receive buffer on the UDP socket. Not the OS socket buffer.

The default buffer size is 3,000 bytes. This should be fine for most GET requests. We have seen cases where something like a VLAN membership variable is gigantic because the switch happens to have thousands of ports.

GETBULK requests can be very large depending on the number of variables, the size of the response and the number of rows requested.

If NOT using buffered SNMP, the same Java buffer is used for every receive so it doesn't really matter how big you make this buffer. If using buffer SNMP, then a buffer is created per receive. In this case you want to be mindful of memory usage and try to size the buffer according to the size of your requests.

Both the SNMPPoll object and the UDPSocket object permit the auto-adjusting of the buffer size if a large packet is received. Currently, NMSCore has the SNMPPoll object adjust the buffer size.

If you browse the discovery.cfg file you will see a section where Network Ferret is pre-sizing processor socket buffers based on experience over the years.

There is a tradeoff between minimizing SNMP requests to a device (asking for large number of rows in GETBULK) and the amount of work the machines have to do recombining the fractured UDP packets. Remember that a UDP packet can hold, usually, a max of a few hundred bytes of data.

There is also the inefficiency of asking for a large number of rows from a device when the table does not exist (or is empty). In this case a large amount of useless data is being sent back. The SNMP infrastructure does permit you, without changing your code logic, to do a large GETBULK but, first, do a single test query to make sure there is something there to get.



# NetConf/REST

These are synchronous protocols. NMSCore has support for these protocols. A Processor subclassed off of `AtiSynchronousProcessor`. A Probe which is the synchronous thread of execution.

These protocols are inherently vendor-specific so you will be using the VSP files to define code to load for a given Host/Probe.

There are also cases where you have a mostly SNMP application but there is this one device that requires NetConf for one specific query. `SNMPTableWalker` has infrastructure to allow you to execute non-SNMP code which is usually executing one of these protocols.

There is support for reading NetConf and REST results from files. This is mostly for debugging. A client site has a problem. NMSCore has logic to allow you to write a result to a file. The client sends you the file. You have the ability to read that file and use the data as if it came from the device.

See the “protocols” package in the example code to see how to use these protocols.



# 8: Utilities

There are various utility classes which may be of use.

## CacheMember

This is in the utils package. AtiSystem has int and String caches. There is much duplicative data in network management. SNMP has its own internal cache which saves the creation/destruction of millions and millions of objects in the course of a large discovery.

## Converter

This is in the utils package. Yes, a strange name. A class that provides various methods for IP addresses (this class existed long before AtiIPAddress), byte array to string conversion, etc.

## DNSLookup

This is in the utils package. Provide filters to skip certain lookups and timeout options. A poorly implemented DNS can severely slow down your application.

## File

Methods for opening/closing files (text and Java serialization) using the Report directory as defined in the config files.

## IFFilter

This is in the Vendor package. This class provides a matching mechanism for interfaces based on various attributes of an interface. The filters that Network Ferret uses are defined in the VSP files but this is not required. Network Ferret uses these filters to exclude certain interfaces during discovery.

## IPAddress

This is in the Core package.

## Queue

This is in the utils package. Network Ferret is 25+ years old. We had to create a Queue class back then.



# Reflection

This is in the utils package. We exposed two methods that the State Machine classes use to find execution methods in other classes.



# 9: Getting Started With Network Ferret

The previous chapters described the NMSCore architecture that is available to you and which Network Ferret itself uses.

This chapter describes the architecture of Network Ferret which is an Agent that runs within the core described above.

## Before Writing Any Code

Before you write any Java code for Network Ferret you should:

- Thoroughly read the User Guide and Domain Model Guide.
- Run Network Ferret enough times on its own that you understand the various configuration parameters and understand the output.
- See the Java source examples which show how to embed Network Ferret.
- Read the remaining chapters if you intend to customize Network Ferret and add discovery functionality.

After doing these things, you can begin writing custom extensions to Network Ferret.

If you have no intention of writing extension, then you do not need to read these chapters at this time. The User Guide and Domain Model Guide will be sufficient.

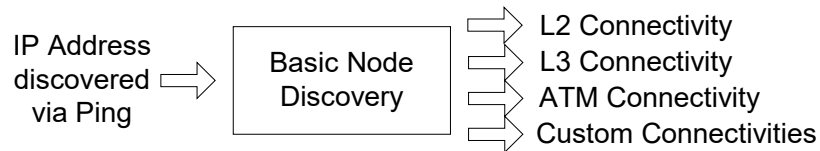
## Network Ferret Architecture

Network Ferret is built on the NMSCore. It is a set of Java code which takes its input from a text-based configuration file and produces a stream of discovery output in the form of Java objects called FactoryShipments. The FactoryShipments go to three optional places: the embedding application, CSV files and Network Ferret's topology database.

Network Ferret is IP-based but it is not completely SNMP-based (NetConf, REST and VMWare API are also used). IP Addresses (called Hosts) are discovered via ICMP ping. Hosts are then subjected to basic discovery. Basic discovery (node inventory) concerns itself

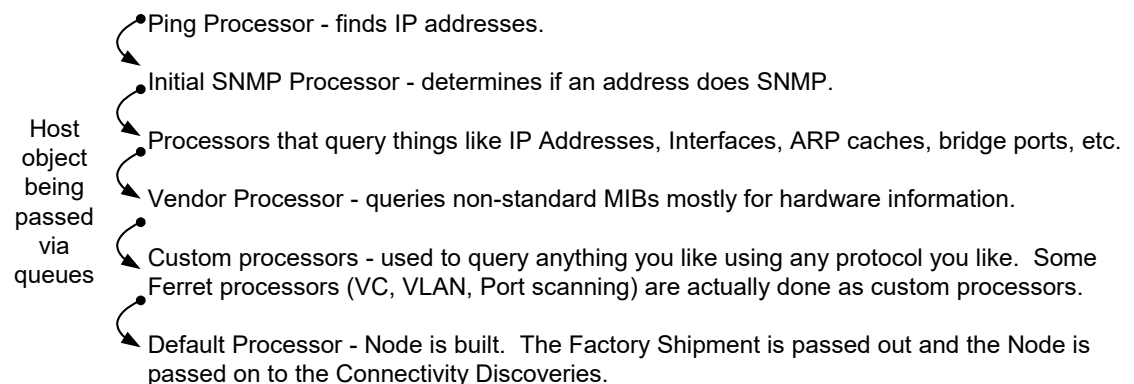


with uncovering information pertaining to a single physical node (which can contain many IP Addresses). Once a Host is finished with basic discovery, it is then passed on to any number of advanced discoveries. Advanced discoveries concern themselves with uncovering information pertaining to connectivity between nodes. Examples of advanced discoveries are layer-2 connectivity and layer-3 connectivity.



Network Ferret is heavily threaded. It makes use of the `AtiSNMPTableProcessor`. Basic discovery is a collection of threads and queues where each thread is responsible for a specific set of data. Each advanced discovery runs in its own thread and is fed from basic discovery by its own queue. All of this threading and queuing prevents bottlenecks. Each of the advanced discoveries is also implemented as a series of threads and queues.

The threading of basic discovery is shown below.



One could ask the question: why not have many threads where each thread does all of the querying for a single node rather than have a thread query all nodes for a specific piece of information.

This is an interesting design question. Both methods were looked at and the threading described above better fit the requirements. Even today (2024), creating 10,000 threads and sockets is not really a good thing.

Using this architecture, Network Ferret is able to discover about 30,000 switches and routers (just the node inventory) in about 60 minutes.



About 350 per minute. This was an actual discovery, not just a calculation based on a smaller discovery.

NMSScore permits both architectures.

## Wire Model

Network Ferret has an internal data structure called the Wire Model. This models connectivity between nodes. Any logic that does advanced discovery (connectivity between nodes) would need to understand how the Wire Model works. It is complex and will not be discussed here until the day comes when a client has a practical application they want to do.

## Customization Options

Note from the diagrams above that there are two main locations where customizations can occur. You can add custom information to a basic node or you can create a custom advanced discovery.

Customizations are not bound to use SNMP or even IP although you will only be told a node exists if it responds to ICMP pings or SNMP.

Customizations are not bound to the domain model as defined in the Network Ferret Domain Model Guide. Customizations are free to add to existing objects or define new ones. The ability to create new objects depends on the embedding program's ability to handle arbitrary data.

Because customizations can add new things to a FactoryShipment, a discreet list of "types" cannot be defined for a FactoryShipment.

## Customizing basic discovery

Create custom basic discovery logic when you want to add information to a Node that is not discovered by the existing basic discovery processors.

Suppose you wanted to model Oracle databases. You could write a custom processor that hits the well-known TNS Listener port and then discovered all of the database instances, tables, users, etc. defined to Oracle on that Node.

Suppose you did not like the way Network Ferret did IP port scanning. You could disable that processor and replace it with your own.



## Customizing advanced discovery

Create custom advanced discovery logic when you want to do something after a Node has been completely discovered. Advanced discovery logic typically involves discovering relationships between nodes such as layer 2 connectivity or layer 3 connectivity.

You can create an advanced discovery to model MPLS paths through a network or perhaps discover connectivity in a TDM multiplexor network.

Modeling connectivity is complex (see Wire Model above).

## Customizing vendor-specific information

In the world of SNMP there are standard MIBs defined that contain important discovery information but not all vendors follow the standards usually because they created their MIBs before the standards were defined and they have not bothered to change them.

In cases like this it is necessary to have logic for each device that does not follow the standard.

Network Ferret provides a framework for implementing vendor-specific support. Existing Network Ferret processors (hardware, VLANs, layer 2, layer 3 and ATM) all use the vendor framework. You are free to add support in these processors for devices Network Ferret does not already support or use the framework in your own customization.

If creating files for a new vendor or modifying an existing vendor, use files with the suffix “vspx”. This way, when a new release comes out, you can easily see the changes you have made.





# 10: Embedding the Discovery Engine

This chapter describes how a programmer would embed Network Ferret in their own application. This chapter assumes a solid understanding of Java.

## Environmental issues

### Platforms and JREs

Network Ferret has been tested with various JREs. You are free to try any JRE although success cannot be guaranteed. Network Ferret does not have any GUI components and does not use any advanced features in Java so there is a very good chance that Network Ferret will run fine in another JRE. The most complicated things Network Ferret does with the environment are to call a few native methods for ICMP and, dynamically load classes and use reflection.

Network Ferret has been tested on Windows 10&11 and RedHat Linux on Intel. We no longer support Solaris (do you youngsters even know what that is!!). It will probably run fine on other platforms given the fact that it was written in Java. Again, success cannot be guaranteed. There is some platform-specific code that would have to be modified. There is a small amount of C code which does the ICMP pings. If you are interested in running Network Ferret on another platform, contact us and we can discuss providing you the ICMP source code.

One client ported the ICMP code to Apple Mac and gave us the binary. It is included in the distribution.

### Java Native Image

Technology exists to “compile” a Java application so no VM is required. This technology is difficult to use with native code, dynamic class loading, serialization and reflection. All of which Network Ferret uses.

We have no desire to try and make this work. Write us a giant check and we'll see.



We can understand the desire for a native image if writing a small service that will be activated hundreds of times a second on some server. Discovery is a heavy process that runs for a relatively long period of time. The efficiencies gained from the compiled image seem not so important to us.

## Third Party Jars

The AMT/java/ext directory contains a number of jars that need to be in your classpath. See amtroot.cmd to see how we set this up for running standalone.

## Working with the configuration files

The format of the configuration files and the definition of each parameter are defined in the various .cfg files in the config directory.

It is your design option to expose or not expose these files to your users. Even a combination of both is feasible. You could create a GUI for novice users which only exposes a few key parameters and provide an advanced option for expert users that either exposes all parameters in a GUI or simply points them to the file and their favorite editor.

# Running discovery programmatically

## Running from within a Java program

See the com.logikos.discovery.examples package in the src directory for an example that shows how to embed Network Ferret.

To run Network Ferret from within your own Java program you will need to do the following:

### Set up the CLASSPATH

Make sure discovery.jar and nmscore.jar is in your classpath.

Look at the discovery command file in the bin directory for an example of setting up the classpath. Also, you should note the parameters that are passed to the VM. These are the parameters that Network Ferret has been tested with. If you choose to modify these parameters for your own application or use others, please be aware that these may have an effect on Network Ferret.

An important note about using Properties. The NF code will contain the same defaults as found in shipped config files so you do not have to supply these parameters unless you are changing the default. You,



in fact, should remove them so that future changes to defaults does not require you to change your code.

## Running as a standalone process

Running Network Ferret as a standalone process is simply a matter of following the example in the discovery command file in the bin directory. It shows the executable to start and the parameters that need to be passed.

Note that there are two types of parameters. There are parameters to the java VM and parameters to Network Ferret itself. The parameters to the Java VM are all of the parameters that appear up to and including `com.logikos.discovery.core.AtiIPDDiscovery` on the command line. The parameters after the class name are passed on to Network Ferret's `main()` as arguments.

## Handling discovery output

This section discusses handling discovery output if you are running Network Ferret from within your own Java application. If you are running Network Ferret as a standalone process, then the output will appear in the report directory as CSV files.

## Registering with Network Ferret

Your application must implement the Java Interface called `AtiIPDExternalInterface` in order to receive any information from Network Ferret. This interface extends the `NMSCore` interface `AtiLoggingInterface`.

```
public class myClass extends Object implements AtiIPDExternalInterface
```

and implement the following methods:

```
public String getName();
public void basicDiscoveryComplete(HashMap stats);
public void discoveryComplete(boolean abnormalTermination);
public void shipment(AtiFactoryShipment aShipment);
public void pingSpecStarted(AtiIPDPingSpec spec);
public void pingSpecCompleted(AtiIPDPingSpec spec, String why);
public void debugMessage(String address, String message);
public void errorMessage(String address, String message, Exception
exception);
public void progressMessage(String address, String message);
```

`getName()` simply returns a `String` which Network Ferret uses as a tag in the log. The other methods are described later.



To register the interface you need to do the following:

```
AtiIPDDiscovery d = new AtiIPDDiscovery();  
d.registerExternalInterface(yourInstanceOfTheInterface);
```

You should register your interface before you call the execute() method. Otherwise, you may miss some messages. It is acceptable to not register an interface. If you don't, discovery will still run and the logs and reports will be created but your program will not receive any callbacks.

## Receiving progress and debug messages

The progressMessage() and debugMessage() methods will be called when Network Ferret generates a progress or debug message for the log (the message will still be written to the log).

There is no return value. You are free to do anything you like with the parameters including nothing. By the time your method is called, the message has already been written to the log. Network Ferret will make no use of the message after your method is called.

Note that the message parameter in the progress method contains the localized version of the message. The message written to the log will be in English. Also, the message parameter will only contain the message whereas the log will contain other information such as the timestamp, function involved, address, etc.

The basicDiscoveryComplete(Hashtable stats) method will be called after basic discovery has completed. You don't necessarily have to do anything with this method. If no advanced extensions are configured to run or if this is a very small network, the discoveryComplete() method will be called shortly after this one. The stats Hashtable contains various bits of information about skipped subnets, addresses, timeouts, etc.

The discoveryComplete(boolean abnormalTermination) method will be called after discovery has completed and cleaned up all of its threads. You don't necessarily have to do anything with this method as your call to execute() will also return when discovery has completed. You can use either discoveryComplete() or the return of execute() as a signal to do your own cleanup.

## Receiving error messages

The errorMessage(String address, String message, Exception exception) method will be called when Network Ferret generates an error message for the log (the message will still be written to the log).



There is no return value. You are free to do anything you like with the String including nothing. By the time your method is called, the message has already been written to the log. Network Ferret will make no use of the message after your method is called.

Note that the message parameter in this method contains the localized version of the message. The message written to the log will be in English. Also, the message parameter will only contain the message whereas the log will contain other information such as the timestamp, function involved, address, etc. Not all error messages have been localized. A non-localized error message will begin with INT: . The INT stands for Internal.

Most times exception will be NULL. Exception will only have a value when a Java exception generated the error. Network Ferret uses the exception to generate its exceptions.txt file.

## **Address parameters in messages**

The address parameter in the debug(), progress() and error() methods should NEVER be NULL. Address may have the value of “noAddr”.

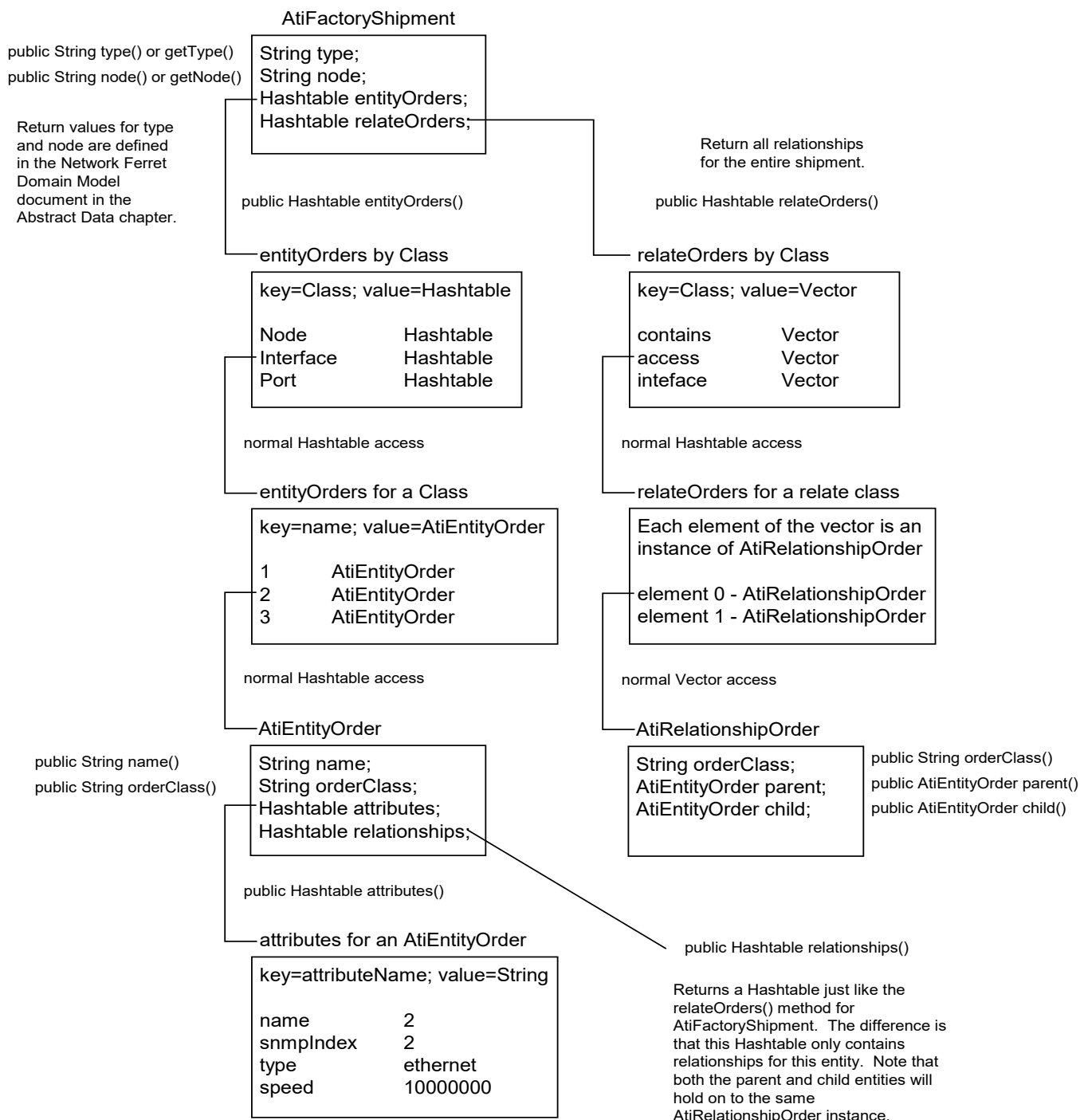
## **Receiving domain information**

Network Ferret produces output using an object called an AtiFactoryShipment. This section will only describe accessing the shipment. For information on the contents of a shipment, see the Network Ferret Domain Model document, especially the chapter regarding Abstract Data Output.

Network Ferret does not provide its own Java domain model (i.e. class implementations) for discovery data. This is because each application has different needs for the discovery data and therefore trying to create one domain model to suit all applications is virtually impossible.



Below is a diagram which shows the layout of an AtiFactoryShipment.  
**NOTE: Hashtable has been replaced with ConcurrentHashMap and Vector has been replaced with ArrayList.**



**See the javadoc for a complete listing of methods available.**

**All of the entity type names, relationship type names and attribute names are defined by constants in the Interface `com.logikos.discovery.factory.AtiFactoryNames`. The source is included in the `java/src` directory.**



The method below gives an example of accessing a shipment. This method uses a few methods that are not included in the diagram above. These are a few convenience methods that take a little of the pain away from the Hashtable manipulation. This is the actual method used to write the raw.csv file in the reports directory.

```
public void writeRawDump(BufferedWriter file, AtiFactoryShipment shipment)
{
    if (null == file)
        return;
    // Write out the entity orders
    // Iterate through the Hashtable for each class
    for (Enumeration e = shipment.entityClasses(); e.hasMoreElements(); ) {
        String eClass = (String)e.nextElement();
        Hashtable eOrders = shipment.entityOrders(eClass);
        for (Enumeration k = eOrders.keys(); k.hasMoreElements(); ) {
            String id = (String)k.nextElement();
            AtiEntityOrder eOrder = (AtiEntityOrder)eOrders.get(id);
            // toString() for Hashtable provides a nicely formatted String.
            // No need to iterate through the attributes ourselves.
            String attrs = "" + eOrder.attributes();
            attrs = attrs.replace(',', ' '); // commas will screw up the CSV file

            try {
                file.write(shipment.node() + "," + eClass + "," + id + "," + attrs);
                file.newLine();
            } catch (Exception ex) {}
        }
    }

    // Write out the relate orders
    // Iterate through the Vector for each class
    for (Enumeration r = shipment.relateClasses(); r.hasMoreElements(); ) {
        String rClass = (String)r.nextElement();
        Vector rOrders = shipment.relateOrders(rClass);
        for (int index = 0; index < rOrders.size(); index++) {
            AtiRelationshipOrder rOrder =
(AtiRelationshipOrder)rOrders.elementAt(index);

            try {
                file.write(shipment.node() + ",Z-" + rClass + "," +
rOrder.parent().orderClass() + "," + rOrder.parent().name() +
", " +
rOrder.child().orderClass() + "," + rOrder.child().name()
);
                file.newLine();
            } catch (Exception ex) {}
        }
    }
}
```

## Error handling

Network Ferret wraps the calls to your methods in a general exception handler. If your code does generate an exception, an error message will be written to the Network Ferret log.

Note that an exception in `errorMessage()` will not produce another error message. This runs the risk of creating an endless loop. An exception in `errorMessage` will produce a debug message to the log.





# Controlling a running discovery

This section discusses controlling Network Ferret if you are running Network Ferret from within your own Java application.

Note that we do not know of anyone who has actually used this feature.

## Pausing discovery

Network Ferret is paused by sending the `pause()` message to the instance of `AtilPDDiscovery`.

The `pause` method will return immediately. Network Ferret is composed of many threads and each will pause in its own timeframe. Network Ferret wants to finish any outstanding requests before pausing so as not to throw devices away due to timeouts. So the time it takes to pause can be as long as the longest timeout defined in the configuration file.

You will not receive any indication as to when Network Ferret has actually paused.

Sending a `pause()` message while discovery is paused has no adverse effects.

## Resuming discovery

A paused Network Ferret is resumed by sending the `resume()` message to the instance of `AtilPDDiscovery`.

The `resume` method will return immediately and all of the paused threads will immediately resume execution. Sending a `resume` message() to a Network Ferret that is not paused will have no adverse effects.

## Stopping discovery

A running Network Ferret is stopped by sending the `stopNow()` or `stop()` messages to the instance of `AtilPDDiscovery`.

The `stopNow()` method sets a flag which all processors and agents look for at various points in their logic. There is no attempt to do anything orderly.

The `stop()` method calls `pause()`, `stopNow()` and then `resume()` which attempts to shutdown in a more orderly fashion.



Stopping Network Ferret is not something you would normally do. It usually only done when someone realizes they are discovering much more than they thought they were.

Discovery CANNOT be stopped until it has finished starting. Stopping it while it is starting produces chaos! Check the method `discovery.finishedStarting()`. `Stop()` and `stopNow()` will do nothing if discovery has not finished starting.

## Running Concurrent Discoveries

Multiple discoveries can be run concurrently.

The only restriction is that you must define output locations in such a way that the discoveries do not try to write to the same file. There are two ways to accomplish this.

Use the config file parameters `IPDReportUniqueNames` and `IPDLogUniqueNames`. Setting these to true causes Network Ferret to generate a subdirectory for each discovery run. The directory name will be the Java millisecond timestamp.

Alternatively, provide different values for `IPDReportRoot` and `IPDLogRoot` for each discovery run.

## Querying Discovery For Progress

The `ProgressState` object is accessible via `discovery.progress()`. The state is updated based on the config parameter `IPDDiscoveryStatus` although the progress of pinging is updated in real time.

See the JavaDoc for specific methods and public fields. It would have been optimal if the calling program could update the progress when it wanted but because of how Network Ferret manages certain variables, only the internal Network Ferret threads have the ability to force a progress update.

Progress is based on things to do rather than time since it is unknown how long any device will take to discover. A device could take 5 seconds or 5 minutes or 5 hours.

The progress is broken up into four sections. Ping, basic discovery, extensions and wire model extensions.

For basic discovery the percentage complete is based on the number of hosts passed from ping (minus timeouts along the way and minus addresses belonging to multi-IP nodes such as routers) along with the



total number of processors (interface, bridge port, entity MIB, etc.). Again, this is based on the hosts to be processed by each processor and not based on any time measurement.

For the extensions, each extension is measured similarly to basic discovery. Note that L2 and L3 need to wait on Connectivity Discovery to complete so they will read 0% until then. Connectivity Discovery tends to always read 100% because it processes hosts as they arrive from Basic Discovery and does so fairly quickly. The extensions can't predict how many hosts they will have to process based on what Ping has found. They can only decide when Basic Discovery is done with a host whether or not they need to process it.

This is still a work in progress.

## Providing Security Information

See the Credentials chapter. Credentials are part of the core infrastructure and not specific to Network Ferret.

## Debugging while using Network Ferret

This section is only important if you are running Network Ferret from within your own Java application.

Minimally, Network Ferret creates 20-30 threads while running. If you are doing synchronous discoveries such as Port, NetConf or VMWare, hundreds of threads can be created. Every thread in Network Ferret is given a name which generally matches the tags found in the log file. You should be aware of this when debugging your own applications.



# 11: Extending Basic Discovery

There are two ways to extend basic discovery. You can use an `SNMPStateMachine` or an `SNMPTableWalker`. An example of each is in `com.logikos.discovery.examples`. `SNMPBasicExtension` and `SNMPBasicExtensionStateMachine`.

Both can be made into vendor-specific logic. See the class comments.

For purposes of minimizing the changes to existing code, any state machine class that Network Ferret will run **MUST** have its class name end with `StateMachine`.

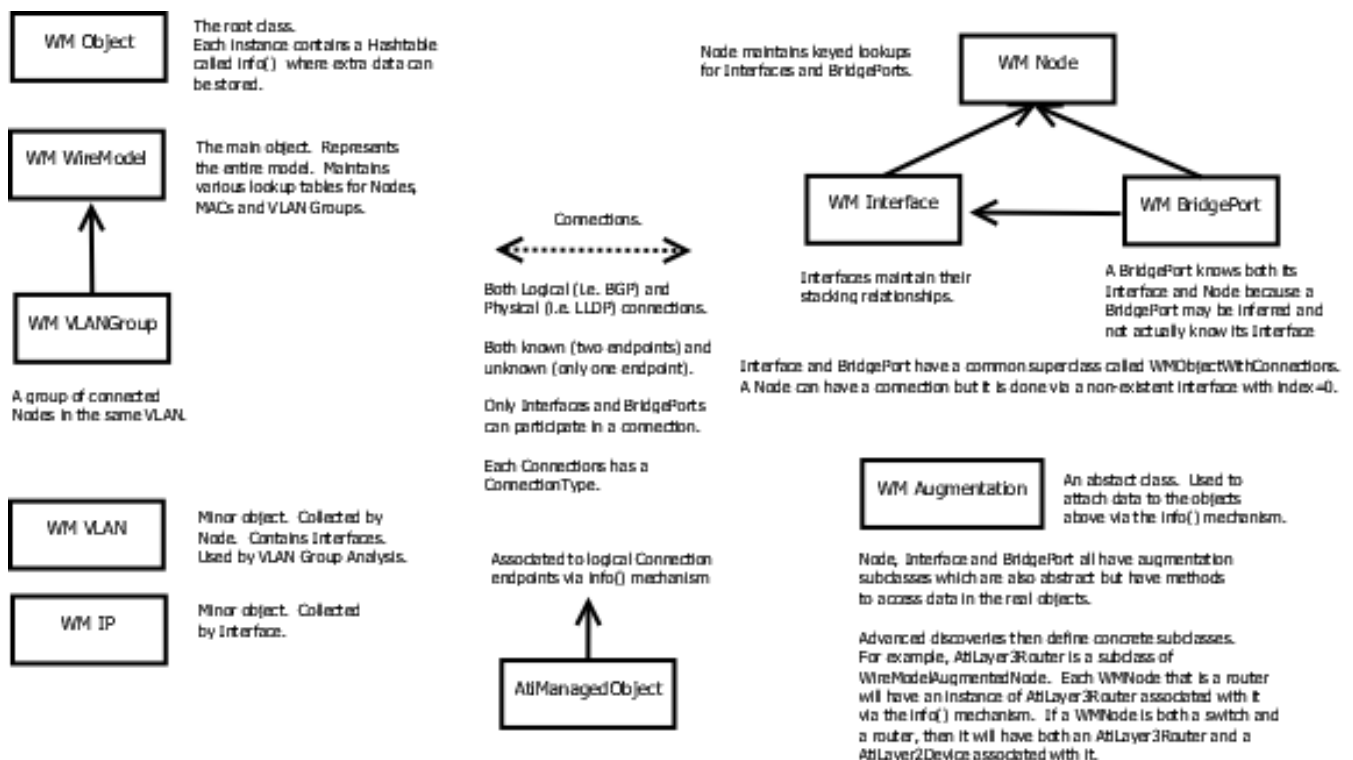
Note that only a single instance of your `StateMachine` class is created. You **CANNOT** store host-specific information in your class. This is different from vendor-specific code where each host gets a new instance of the vendor class. Please read the class comment in the `StateMachine` example.



# 12: The Wire Model

Version 11.0 introduces the concept of a Wire Model. The purpose of the wire model is to provide a common model for advanced discoveries where connections between Nodes can be made.

In previous versions, each advanced discovery had its own data structures. Sometimes discovery information was replicated in the various discoveries. Sometimes a given discovery was lacking information that another had.



## Design Considerations

One goal was to minimize the disruption/change to existing NF code. This was the purpose of the **WMAugmentation** classes.

Each advanced discovery had a representation of a Node and an Interface/Port. Some of the information in these objects was copied from basic discovery. The same data is now contained in **WMNode/Interface/Port**. Some information in these objects was



specific to the particular advanced discovery such as MPLS data for L3.

By removing the common data/code from these classes and then allowing them to be associated with their wire model counterpart, we were able to retain much of the coding logic in the advanced discoveries with respect to protocol specific processing but remove/consolidate/cleanup the processing that was similar across the discoveries.

The other advantage of the WMAugmentation classes is that one developer can work on L2 code and another on L3 without interfering with each other. If we had combined all of the fields into one giant WMNode or WMInterface object, team coding would be more problematic.

Attaching AtiManagedObjects to Connection end points also minimized the changes to protocol specific logic and to the creation of the factory shipments. For logical connections such as BGP, OSPF, etc. the data was always maintained in subclasses of AtiManagedObject. These objects were then simply added to the shipments – such as BGPPeer or OSPFNeighbor. The shipment remains the same but the technique of creating the shipment from these objects is now common across the various advanced discoveries.

## Combining Discoveries

One design goal was the ability to “knit together” two or more discoveries.

## Serialization

The wire model is serializable and is written to the report directory at the end of discovery. To minimize the possibility of disaster (serializing everything from Basic Discovery), a new type of AtiManagedObject was created. AtiIPDSerializableManagedObject. This class is subclassed off of AtiMangedObject. Only classes that need to be serialized were moved to this tree. Classes such as AtiBGPPeer and AtiSISAdjacency.

If something in the WireModel or something in the advanced discovery classes is holding on to any other ManagedObject types, they will not get serialized.

Another possibility would be allow AtiManagedObject to implement Serializable, override the serialization methods to ask the MO if it is allowed to be serialized and either allow the default serialization



behavior to occur or end the method. The downside of this is that every MO subclass would have to be edited to include the UUID field so the compiler does not complain.

## Size of the Wire Model

In early testing we have found that the wire model can be quite large and slow to serialize. We experienced 1 hour with one large network. A single device had 24,000 interfaces.

We have made as many fields as possible *transient*. We also have implemented logic where the wire model purges interfaces without any interesting information before the serialization occurs.

## Knitting

There is a subclass of WireModel called WireModelGroup. It is a collection of WireModels. It overrides certain methods in WireModel to make the group appear as a single WireModel to all callers.

Advanced discovery connection logic looks for an existing connection. If none is found, as would happen when running a regular discovery, a connection is created. That connection can either be known (we found the other end) or unknown (we did NOT find the other end).

If an existing unknown connection is found, as would happen when knitting together multiple discoveries, and we find the other end, the unknown connection (and possibly one on the other end) is replaced with a known connection.

Connection IDs for knitted connections start at 1,000,000 so it is easy to tell if a connection was from a single discovery or created when discoveries were combined.

## Performing a Knit

The code below shows how to perform a knitting operation. Where you stored/retrieved your wire models is up to you.

```
import com.logikos.topology.wireModel.*;

WireModelGroup group = new WireModelGroup();
group.setName("knitting test");
group.addWireModel(wm1);
group.addWireModel(wm2);

AtiIPDDiscovery d;

String rootDir = "c:\\amt";
```



```
String configDir = "config";
String configFile = "home.cfg";

d = new AtiIPDDiscovery();
d.setWireModel(group);

d.execute(rootDir, configDir, configFile);

System.out.println("DONE KNITTING TEST");
```

Discovery will see that the wire model has been set and perform the knitting operation rather than running discovery. It was necessary to perform the operation in the context of running discovery in order to use the config/reporting/logging infrastructure. Advanced discovery results will be reported as via the API and in the files, The same as with a regular discovery.

Use WM\_ReportKnittedConnectionsOnly - true in the config file to only see the new connections that were made.

## What Is and Is Not Analyzed on a Knit

Certain advanced discovery analysis cannot be performed during a knitting operation. For example, L3 will not query routing tables and L2 will not query the bridge forwarding tables. No new querying is done.

For L2, it was decided that keeping the data from the bridge forwarding tables in the wire model was too much. So this analysis is only done during a running discovery.

In general, specific protocol data such as LLDP or CDP will be analyzed in a knit. "Guessing" types of operations such as analyzing bridge forwarding table, will not.

## Accessing The Wire Model

If anyone would like to programmatically access the Wire Model, let us know and we will put together some examples. In theory





# 13: Creating An Advanced Discovery

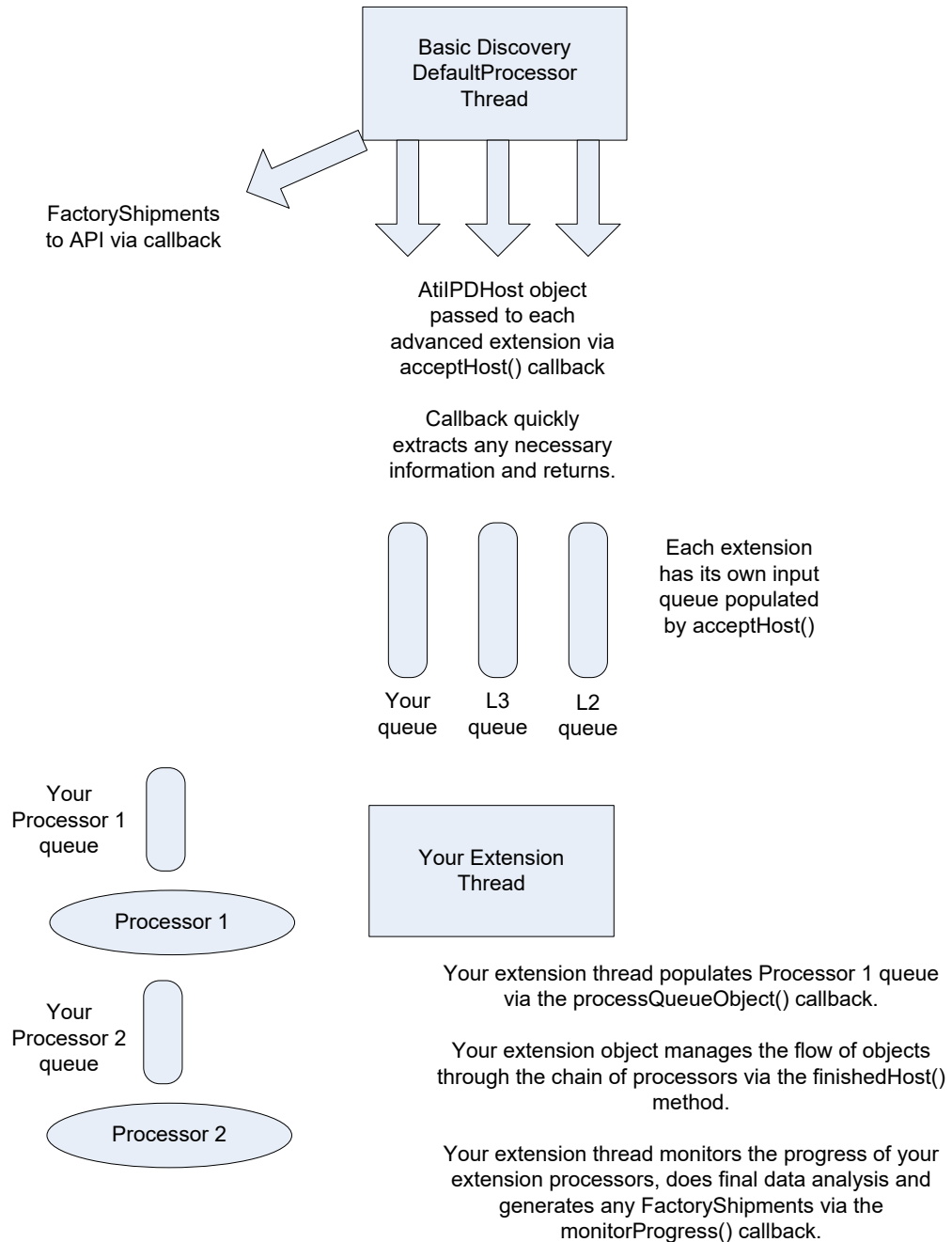
An advanced discovery extension is different from a basic discovery extension in several ways.

An advanced extension will subclass off a different class although an advanced extension will generally create its own chain of processors to do the actual work.

An advanced extension is complex and will not be documented here until there is a client who has a real need to write their own.



# Advanced Discovery Overview



# 14: Localization

Network Ferret progress and most error messages have been set up so they can be localized. Network Ferret ships with a default set of messages in English.

All debug messages will be generated in English. Non-localized error messages will begin with INT: . Localized progress and error messages will be passed out via the API. Network Ferret logs will always be written in English.

## Setting the Locale

See the MsgAgent example in `com.logikos.core.examples.message`. This example shows how to create messages for your own agent but it also shows how to translate NMSCore messages. The same technique would be used for Network Ferret.

NMSCore has messages and NetworkFerret has messages. Both would need to have translations created.

