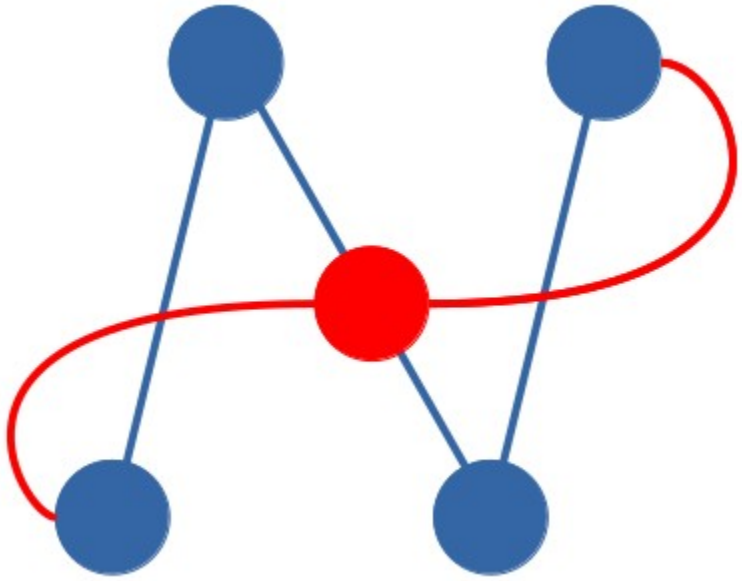


NeuPaths SDK

Version 1.5.0

Table of Contents

1 Introduction

The *NeuPaths* SDK provides a framework for massively parallel, distributed, asynchronous processing modeled after the human nervous system. *NeuPaths* is an API and runtime environment for a new programming paradigm called *Cellular Programming*. Cellular programs operate like the human mind, where autonomous cells work together to solve complex problems. *Cells* are interconnected by *Synapses* and organized into clusters (see figures 1 and 2.) The cells in a cluster process *Stimuli* that traverse the synapses, producing new stimuli in response. Clusters are also interconnected by synapses and together form a *System*. A cellular system forms an undirected multigraph; cells can be connected to one another multiple times, and all synapses are bidirectional.

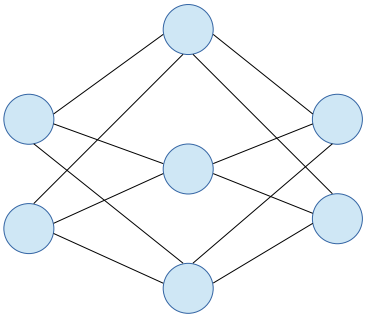


Figure 1: Complex Cluster Example

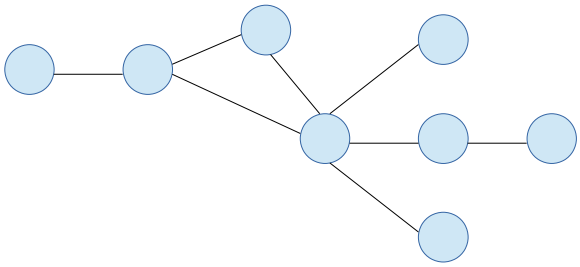


Figure 2: Simple Cluster Example

Each cell has a *Nucleus* and a collection of *Activators*. The cell nucleus routes stimuli to interested activators and neighboring cells. Activators process stimuli on their *Receptors* and advertise new stimuli on their *Transmitters* (see figure 3.) Activators express interest in stimuli using *Subscriptions*. Each activator has its own set of receptors, transmitters and subscriptions. The names of receptors and transmitters are not required to be unique among a cell's activators; activators can share receptors and transmitters as long as they agree on the stimulus type.

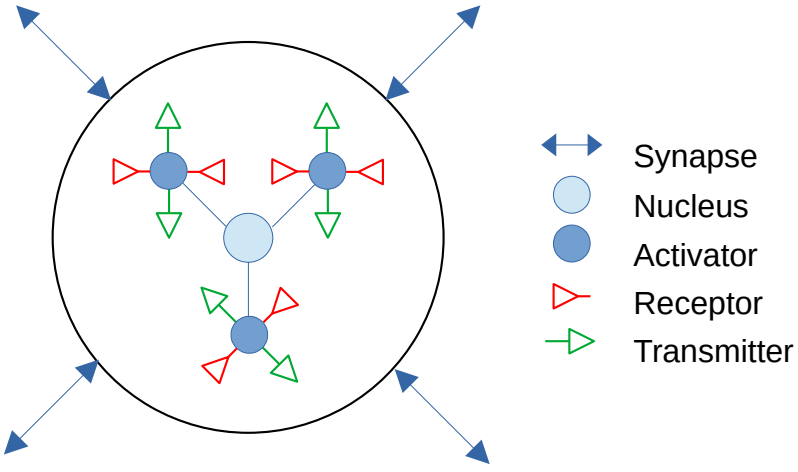


Figure 3: Anatomy of a Cell

The *NeuPaths* SDK is written in pure Java SE, making it possible to achieve "write once, run anywhere." It provides the capability to solve any computational problem through its flexibility:

- A computational problem can be decomposed into cell clusters at any granularity.
- A computational problem can be distributed at any granularity.
- Data processing can be deterministic or non-deterministic by design.
- The computational model can be static or dynamic.
- The framework supports user-defined complex data types.
- A cellular system can interface with, or be embedded within, a conventional sequential/linear program.

2 Getting Started with NeuPaths

A *NeuPaths* system is an asynchronous, distributed data processor in which each cell autonomously consumes and/or transforms data. The activators in each cell request stimuli, which can arrive in any order. When all stimuli have arrived, an activator evaluates the stimuli and may produce new stimuli.

Hereafter, the term *system* will represent a cluster or an entire system, as a system could be viewed as one very large cluster.

To create a cellular system, one must first identify the data, its producers and its consumers. This is known as a data-flow analysis. It is a design process that results in a *Data Flow Model*, which is a high-level description and/or depiction of the cells, stimuli and dependencies in the system.

The next step is to consider cell interconnections. The cellular system may be partitioned into domains that compartmentalize functionality, services and stimuli transmission. This step produces the *Cluster Model*, which describes and/or depicts the "connectedness" of the cellular system. The structure of a cellular system can greatly influence its performance and resilience.

The final step is to identify where each cell resides in the system. The system may be contained within a single host, on multiple nodes in a single intranet, or spread across nodes in multiple interconnected data networks. These details are captured in the *Network Model*. This model influences the synapse names that are used in the system.

It is likely that designing a cellular system will require iteration over these steps. For example, security and performance requirements may impact the Network model, which will then influence the Cluster model. Algorithm refinement may result in decomposition of cells into new clusters, thereby impacting all models.

2.1 Data Flow Model

A data-flow analysis should answer the following questions:

- What kinds of data are needed? (Stimuli)
- Who will consume and/or transform the data? (Cells and Subscriptions)
- How will the data be consumed/transformed? (Receptors, Transmitters and Activators)

2.1.1 Stimuli

All stimuli are specializations of the neupaths.api.Stimulus class. Each specialization constitutes a unique user-defined stimulus type. Receptors and transmitters perform stimulus type checking upon receipt and transmission respectively. Refer to the *NeuPaths* API documentation for neupaths.api.Stimulus class specialization instructions.

The *NeuPaths* API includes a suite of primitive stimulus types in the neupaths.stim package:

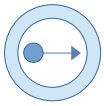
BooleanStimulus:	Conveys a Java boolean primitive value.
ByteStimulus:	Conveys a Java byte primitive value.
CharacterStimulus:	Conveys a Java char primitive value.
DateStimulus:	Conveys a java.util.Date value.
DoubleStimulus:	Conveys a java double primitive value.
FloatStimulus:	Conveys a java float primitive value.
IntegerStimulus:	Conveys a java int primitive value.
LongStimulus:	Conveys a java long primitive value.
ShortStimulus:	Conveys a java short primitive value.
SignalStimulus:	An alias for BooleanStimulus that uses true for an On signal and false for an Off signal.
StringStimulus:	Conveys a java String value.
TimeEventStimulus:	An alias for DateStimulus that formats the Date string value as “mm/dd/yy hh:mm:ss.nnnnnnnnn” (time to nanosecond resolution.)

2.1.2 Cells

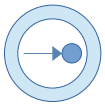
Cells are interconnected autonomous agents that process and route stimuli. The cell nucleus examines each arriving stimulus, checks for matching subscriptions, and routes the stimulus to interested activators and neighbors. There are six types of cells, each used as-is without specialization:

Bridge cells route stimuli between domains. Stimuli transmission is limited to subscription domain, but bridge cells use synapses and subscriptions in two different domains to facilitate a domain bridge. Bridge cells do not have activators, but they support multiple subscriptions for bridging. There is no imposed limit for the number of neighboring cells they can be connected with. Refer to the *NeuPaths* API documentation for neupaths.api.BridgeCell details.

Event cells process log events from a cell domain. Event data can be output to a PrintStream (e.g. System.out) or provided to a specialization of the neupaths.api.EventActivator class. Refer to the *NeuPaths* API documentation for neupaths.api.EventCell and neupaths.api.EventActivator details.



Extractor cells receive stimuli from a system. They do not contain activators, but they do publish a single subscription. Extraction blocks until the desired stimulus is received. There is no imposed limit for the number of neighboring cells they can be connected with. Refer to the *NeuPaths* API documentation for neupaths.api.ExtractorCell details.



Injector cells insert stimuli into a system. They do not contain activators and do not subscribe to stimuli. They emit stimuli on a single user-specified transmitter. There is no imposed limit for the number of neighboring cells they can be connected with. Refer to the *NeuPaths* API documentation for neupaths.api.InjectorCell details.

Logic cells consume and transform stimuli using their activators. There is no imposed limit for the number of neighboring cells they can be connected with, nor for the number of activators they host. Refer to the *NeuPaths* API documentation for neupaths.api.LogicCell details.

Router cells assist with stimuli transmission in complex systems. While all cell types are capable of routing stimuli, router cells can help with complex, distributed systems. For example, a router cell can be used to consolidate network traffic through a proxy. Router cells do not contain activators and do not subscribe to stimuli. There is no imposed limit for the number of neighboring cells they can be connected with. Refer to the *NeuPaths* API documentation for neupaths.api.RouterCell details.

2.1.3 Subscriptions

Subscriptions advertise a cell's interest in a particular type of stimuli. A subscription declares that all stimuli produced by a specified cell and transmitter be placed on an activator's receptor. The producer's cell and transmitter names may contain regular expressions. At runtime, the producing transmitter and consuming receptor stimuli types must match. Refer to the *NeuPatghs* API documentation for the following classes in the neupaths.api package:

BridgeSubscriptionSpec:	Specification used for Bridge cell subscriptions.
ExtractorSubscriptionSpec:	Specification used for Extractor cell subscriptions.
LogicSubscriptionSpec:	Specification used for standard Logic cell subscriptions.
LogicLoopbackSubscriptionSpec:	Specification used for Logic cell loopback subscriptions. An activator requests stimuli from its own cell (either from itself or a different activator.)
LogicMapSubscriptionSpec:	Specification used for Logic cell map subscriptions. An activator requests that stimuli from a specified transmitter be placed on a receptor with the same (or matching) name.

2.1.4 Receptors

Receptors are named, type-safe receptacles for stimuli. Receptors have two modes of operation: buffered and non-buffered. As the names suggest, the buffered mode maintains a queue of stimuli while the non-buffered mode will contain at most a single stimulus. Refer to the *NeuPaths* API documentation for the following classes in the neupaths.api package:

ReceptorMode:	An enumeration of the receptor modes.
ReceptorSpec:	Specification for a receptor.

2.1.5 Transmitters

Transmitters are named, type-safe emitters of stimuli. Refer to the *NeuPaths* API documentation for details on the neupaths.api.TransmitterSpec class.

2.1.6 Activators

Activators consume and transform stimuli. Everything up to this point has been concerned with transmission and delivery of stimuli. The ultimate goal is to supply stimuli to activators. All activators are specializations of the neupaths.api.Activator class. Activators advertise their receptors, transmitters and subscriptions to the cell, which in turn advertises subscriptions and stimuli to the system. An activator evaluates when all of its desired stimuli arrive. Stimuli can arrive in any order, and if using buffered receptors, they can queue up until the complement of receptors is complete. An activator can use a combination of buffered and non-buffered receptors. Refer to the *NeuPaths* API documentation for neupaths.api.Activator class specialization instructions.

2.2 Cluster Model

Having identified the cells, activators and stimuli needed in the system, the next step is to connect everything together. It is possible that the Cluster model closely resembles the Data Flow model; the Cluster model for a simple system might translate data-flow dependencies directly into connections. More complex systems, however, might need router cells to join clusters or bridge cells to facilitate cross-domain interactions. Router and bridge cells might also be useful as network entry points for distributed cellular systems.

Some things to consider when designing the Cluster model:

- If multiple cells subscribe to the same stimuli, do they need to process the stimuli at roughly the same time? If so, then the stimuli path through the cluster to each cell should have a similar length.
- Do you need to prevent certain stimuli from propagating throughout the system? If so, then one or more domains should be used. Judicious use of domains can prevent extraneous subscription and stimuli traffic in the system. Keep in mind that subscriptions in the global domain will reach every cell in the system. Also note that subscriptions are periodically refreshed, so those same global subscriptions will repeatedly traverse the entire system. Conversely, named domains confine subscriptions within their boundaries.
- How long will stimuli evaluation take? If the algorithm requires a lot of CPU, then perhaps the cell should not be central to the system, where it would have to route stimuli frequently.

2.2.1 Domains

Stimuli can be restricted to a particular domain in a system. Cells can participate in multiple domains by using synapses in each domain. Each synapse communicates in a single domain, and subscriptions are also associated with a single domain. Domains are a way to partition or modularize a system. A domain could be managed by a different software team, provide a particular service or just delineate a cluster.

2.2.2 Subscriptions and Routing

Cells express interest in stimuli using subscriptions. Subscriptions are restricted to a particular domain and will only propagate over synapses in the same domain. The only exception to this rule is for subscriptions in the global domain, which will propagate to all cells in a system. Figure 4 illustrates cell B subscribing in domain D_1 to a stimulus from cell A. The subscription will reach as far as cell D, which partially resides in domain D_1, but will not propagate into domain D_2. We use the following convention to denote subscriptions: S_{xyi}, where S indicates a subscription, *x* is the consuming cell's label, *y* is the producing cell's label and *i* is the subscription domain.

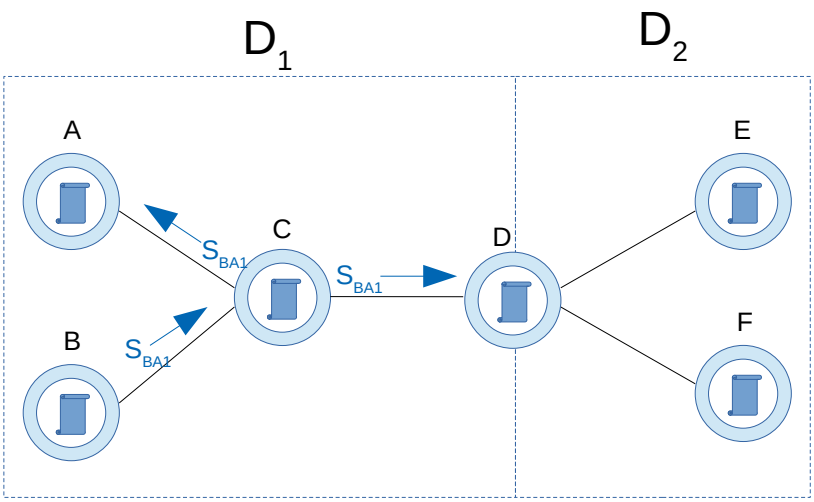


Figure 4: Cell B Subscribes to Cell A in Domain D_1

Each cell that receives a subscription will take note of the synapse over which it arrived. When a transmitter emits a stimulus, the stimulus is *pulled* through the system over the synapses corresponding with the subscription. A subscription tells a cell: "When you see this type of stimulus, send it this way." In this example, a stimulus emitted by cell A would travel to cell C and then to cell B.

What if cell B were interested in stimuli from cell E? Since cell B only resides in domain D_1, a subscription to stimuli from cell E in D_2 would not propagate at all. However, cell B could subscribe to stimuli from cell E in domain D_1. The diagram would look very similar, but stimuli would still not reach cell B because its subscription never made it past cell D.

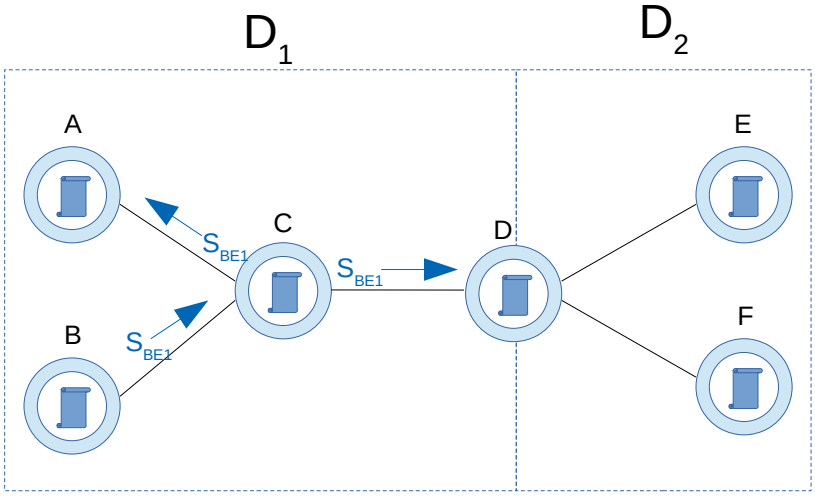


Figure 5: Cell B Subscribes to Cell E in Domain D_1

There are a few ways that stimuli from cell E in domain D_2 could be pulled into domain D_1. The first and most obvious would be if cell D were also interested in cell E's stimuli. If cell D were a logic cell that resided in both domains and subscribed to cell E's stimuli, then cell D's nucleus would receive the stimuli from domain D_2, find a matching subscription in domain D_1 and forward the stimuli over cell D's synapse in domain D_1.

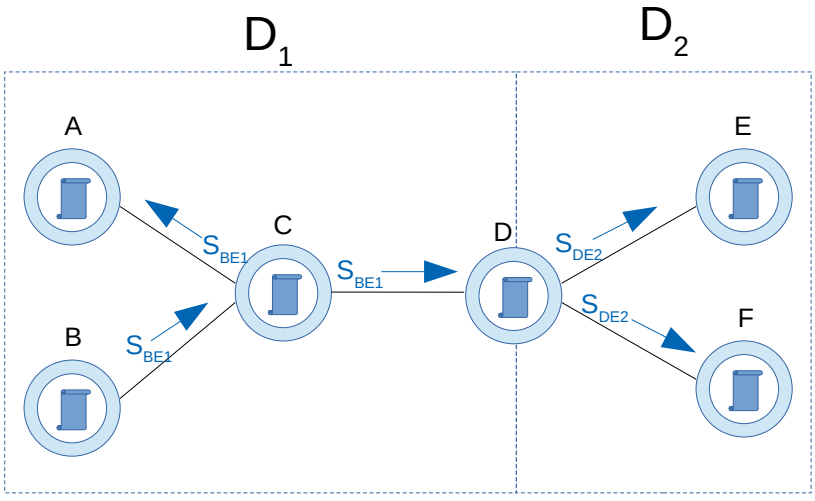


Figure 6: Cell D Subscribes to Cell E in Domain D_2

Another way would be if cell F were interested in stimuli from cell E in domain D_2. A stimulus would travel through cell D on its way to cell F. Cell D would then find the subscription for cell E stimuli in domain D_1 and forward the stimulus.

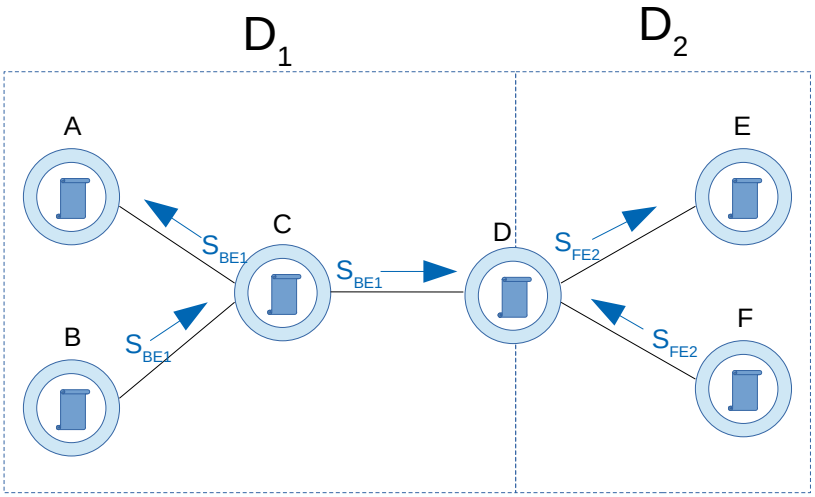


Figure 7: Cell F Subscribes to Cell E in Domain D_2

If there were no tangential interest in domain D_2 for cell E stimuli, the final way to pull stimuli from domain D_2 into domain D_1 would be the use of a bridge cell. A bridge cell maintains a presence in multiple domains and uses subscriptions to pull stimuli across domain boundaries.

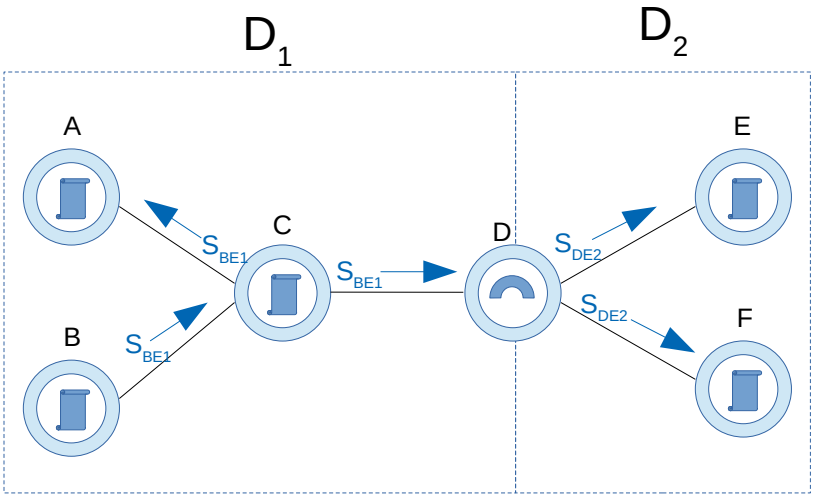


Figure 8: Bridge D Subscribes to Cell E in Domain D_2

2.3 Network Model

The Network model is the most fluid of the three models. In general, none of the API client code (stimulus types, activators, etc.) has to change when a synapse changes. A cell's behavior is independent of its location (unless there are specific hardware requirements such as sensors.) This makes it possible to test a system in whole or part in a test environment that is very different than production without changing any code. It also makes it possible to scale the system to meet resource demands.

The Network model is important though. It specifies where clusters will execute, the protocols to be used, special security concerns and network hardware (switches, routers, firewalls, etc.) The resulting synapse names will reflect all of these factors.

2.3.1 Synapses

Synapses move stimuli throughout a system. They are the glue that holds a system together. They are also an abstraction for the underlying network transport protocols. *NeuPaths* programmers will never use synapses directly. There are no classes to instantiate or specialize. They are created transparently through synapse specifications. Each synapse belongs to a domain. Stimulus subscriptions are also associated with a domain, and subscriptions will only propagate over synapses in the same domain. There is one exception to that rule: Subscriptions in the global domain will propagate across an entire system.

Synapses are specified by string-encoded names with the following format:

scope#*type*#*mode*#*domain*[#*opt1*[#*opt2* … [*optN*]]]

scope	Network or Local
type	Stream, Unicast or Multicast
mode	Listener or Peer
domain	Case-sensitive name or "@" for the global domain.
opt_i	Synapse type-specific options

2.3.1.1 Network Stream Synapses

Network Stream synapses use TCP/IP. They are specified as follows:

Network#Stream#Listener#*domain*#*port*#*host_name*

Network#Stream#Listener#*domain*#*port*#host_*address*

Network#Stream#Listener#*domain*#*port*

Network#Stream#Peer#*domain#port#host_name*

Network#Stream#Peer#*domain#port#*host_*address*

Network#Stream#Peer#*domain#port*

domain	Case-sensitive name or “@” for the global domain.
port	IP port.
host_name	Name of the host. The system will attempt to resolve the host name to an IPv4 or IPv6 address, depending on the system configuration.
host_address	IPv4 or IPv6 address of the host. For Listener synapses, the IP address accepting connections. For Peer addresses, the IP address to contact. Addresses have an optional “4/” or “6/” prefix to denote IPv4 or IPv6 respectively.

These synapses operate in two modes: Listener and Peer. Listener mode waits for Peers to join. When a Peer joins the Listener, a new Peer synapse is created to communicate with the requester.

For Listener synapses, you may omit the IP address, specify “*” or specify “4/*” to use IPv4 INADDR_ANY. You may specify “6/*” to use IPv6 INADDR_ANY.

For Peer synapses, you may omit the IP address, specify “*” or specify “4/*” to use the first valid local IPv4 address. You may specify “6/*” to use the first valid local IPv6 address.

2.3.1.2 Network Unicast Synapses

Network Unicast synapses use UDP/IP. They are specified as follows:

Network#Unicast#Listener#*domain#port#host_name*

Network#Unicast#Listener#domain#port#host_address

Network#Unicast#Listener#*domain#port*

Network#Unicast#Peer#*domain#port#host_name*

Network#Unicast#Peer#*domain#port#*host_*address*

Network#Unicast#Peer#*domain#port*

domain	Case-sensitive name or “@” for the global domain.
port	IP port.
host_name	Name of the host. The system will attempt to resolve the host name to an IPv4 or IPv6 address, depending on the system configuration.
home_address	IPv4 or IPv6 address of the host. For Listener synapses, the IP address accepting connections. For Peer synapses, the IP address to contact. Addresses have an optional “4/” or “6/” prefix to denote IPv4 or IPv6 respectively.

These synapses operate in two modes: Listener and Peer. Listener mode waits for Peers to join. When a Peer joins the Listener, a new Peer synapse is created to communicate with the requester.

For Listener synapses, you may omit the IP address, specify “*” or specify “4/*” to use IPv4 INADDR_ANY. You may specify “6/*” to use IPv6 INADDR_ANY.

For Peer synapses, you may omit the IP address, specify “*” or specify “4/*” to use the first valid local IPv4 address. You may specify “6/*” to use the first valid local IPv6 address.

2.3.1.3 Network Multicast Synapses

Network Multicast synapses use Multicast UDP/IP. They are specified as follows:

Network#Multicast#Peer#*domain#port#multicast_group*

domain	Case-sensitive name or “@” for the global domain.
port	IP port for the multicast group.
multicast_group	IPv4 or IPv6 address of the multicast group.

2.3.1.4 Local Stream Synapses

Local Stream synapses use Unix-Domain streams. They are specified as follows:

Local#Stream#Listener#*domain#filesystem_path*

Local#Stream#Listener#*domain#filesystem_path*

Local#Stream#Peer#*domain#filesystem_path*

Local#Stream#Peer#*domain#filesystem_path*

domain	Case-sensitive name or “@” for the global domain.
filesystem_path	File system path for the Unix-Domain socket. For Listener synapses, the file system path accepting connections. For Peer synapses, the file system path to contact.

These synapses operate in two modes: Listener and Peer. Listener mode waits for Peers to join. When a Peer joins the Listener, a new Peer synapse is created to communicate with the requester.

3 Transactions

Stimuli generally flow through a system without any identifying information. The only distinctive characteristics are the originator, stimulus type and order of arrival. However, the *NeuPaths* framework also supports transactions, allowing a cell to operate as a service provider. Transaction IDs can be assigned to stimuli via the *NeuPaths* API.

Let’s look at an example of how transactions are processed. In this example, we’ll have an injector that requests a Cluster-wide property value. The ClusterManager cell will process the request using a nested transaction with the PropertyManager. The result will flow back to an extractor. Here’s a combined data flow and cluster model:

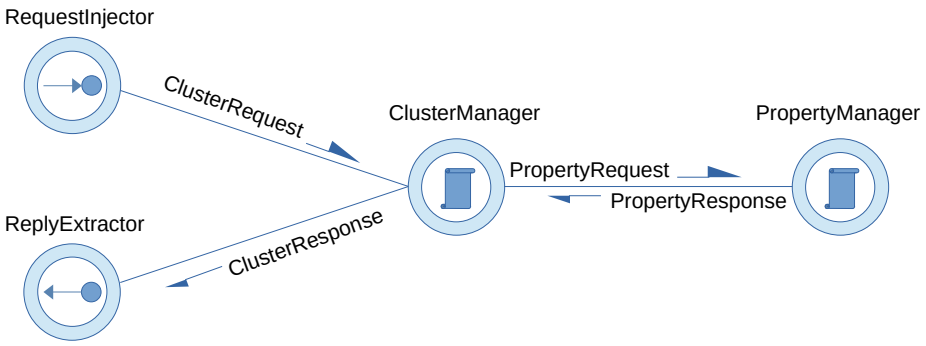


Figure 9: Example Transaction Model

The RequestInjector will initiate the transaction by sending a ClusterRequest stimulus. The neupaths.api.InjectorCell class provides two methods for using a transaction:

- injectAsTransaction – Injects the stimulus as a new transaction and returns the transaction ID.
- InjectWithTransaction – Injects the stimulus as part of the specified transaction.

Excerpt from Main.java

```
InjectorCell reqInj = cluster.getCell("RequestInjector");

UUID transID =
    reqInj.injectAsTransaction(new ClusterRequest("ExampleName"));
```

The ClusterManager cell will have two activators: ClusterRequestActivator and ClusterResponseActivator. The ClusterRequestActivator will create a new transaction tied to the original and transmit a PropertyRequest stimulus. The ClusterResponseActivator will process the PropertyResponse stimulus and transmit the ClusterReponse stimulus under the original transaction. Note that when an activator creates a new transaction, it applies to the entire cell; all activators in a cell have access to the active transactions.

The eupaths.api.Activator class has several methods for managing transactions:

- createTransaction – There are two flavors of this method: one with and one without a response transaction ID. This example uses the response transaction ID to chain together related transactions.
- setStimulus – There are two flavors of this method as well: one with and one without a transaction ID. This example uses the transaction ID to tag the stimulus as part of a transaction.
- isTransactionOriginator – An activator may receive stimuli for responses to transactions originated by other cells, depending on the subscriptions in use. This method indicates if the calling cell originated the specified transaction.
- getResponseTransactionID – Retrieves the transaction ID that should be used in a reply. This would be the value supplied to createTransaction to chain together two transactions.
- terminateTransaction – Releases resources associated with a transaction originated by the calling cell. Use this method when all processing for the transaction has completed.

ClusterRequestActivator.java

```
@Override
protected void evaluate ()
{
  ClusterRequest clusterReq = getStimulus("ClusterRequest");

  // Create transaction tied to original transaction
  UUID transactionID =
      createTransaction(clusterReq.getInstanceID(),
                        clusterReq.getTransactionID());

  // Send PropertyRequest with new transaction
  setStimulus("PropertyRequest",
              new PropertyRequest(clusterReq.propertyName),
              transactionID);
}
```

ClusterResponseActivator.java

```
@Override
protected void evaluate ()
{
  PropertyResponse propertyResp = getStimulus("PropertyResponse");

  if (isTransactionOriginator(propertyResp.getTransactionID()))
  {
    UUID respTransactionID =
        getResponseTransactionID(propertyResp.getTransactionID());

    setStimulus("ClusterResponse",
                new ClusterResponse(propertyResp.propertyValue),
                respTransactionID);

    terminateTransaction(propertyResp.getTransactionID());
  }
}
```

The PropertyManager cell will use a single activator named PropertyRequestActivator. This activator will retrieve the property value and send a response as part of the transaction created by the ClusterRequestActivator.

PropertyRequestActivator.java

```
@Override
protected void evaluate ()
{
  PropertyRequest propertyReq = getStimulus("PropertyRequest");

  // Copy request's transaction ID to maintain transaction chain
  setStimulus("PropertyResponse",
              new PropertyResponse(
                  getProperty(propertyReq.propertyName)),
              propertyReq.getTransactionID());
}
```

Finally, the ReplyExtractor will wait for a reply to the original transaction. The extractFromTransaction method in neupaths.api.ExtractorCell waits for a stimulus tagged with the desired transaction ID.

Excerpt from Main.java

```
ExtractorCell respExtr = cluster.getCell("ReplyExtractor");

ClusterResponse resp = respExtr.extractFromTransaction(transID);
```

4 Learning by Example

The best way to learn cellular programming is through an example, and the customary example would be “Hello World.” Cellular programming is a different paradigm though, so we will create “World, Hello.” The WorldHello program will consist of cells representing different cities in the world, each of which responds with a salutation in its native language when hailed.

4.1 WorldHello Data Flow Model

The goal of WorldHello is to have all cities respond to a global hailing. This implies that each city will listen for the same signal. A logical solution would be to have a single entity hailing the cities and listening for their salutations. For the sake of simplicity, the WorldHello program will be a conventional Java program that creates a simple cellular system. The following data flow model is one possible solution (see figure 10.) The directional lines represent subscriptions and the circles represent cells. The injector is used to inject the Hello stimulus, and the extractor is used to extract each of the Salutation stimuli.

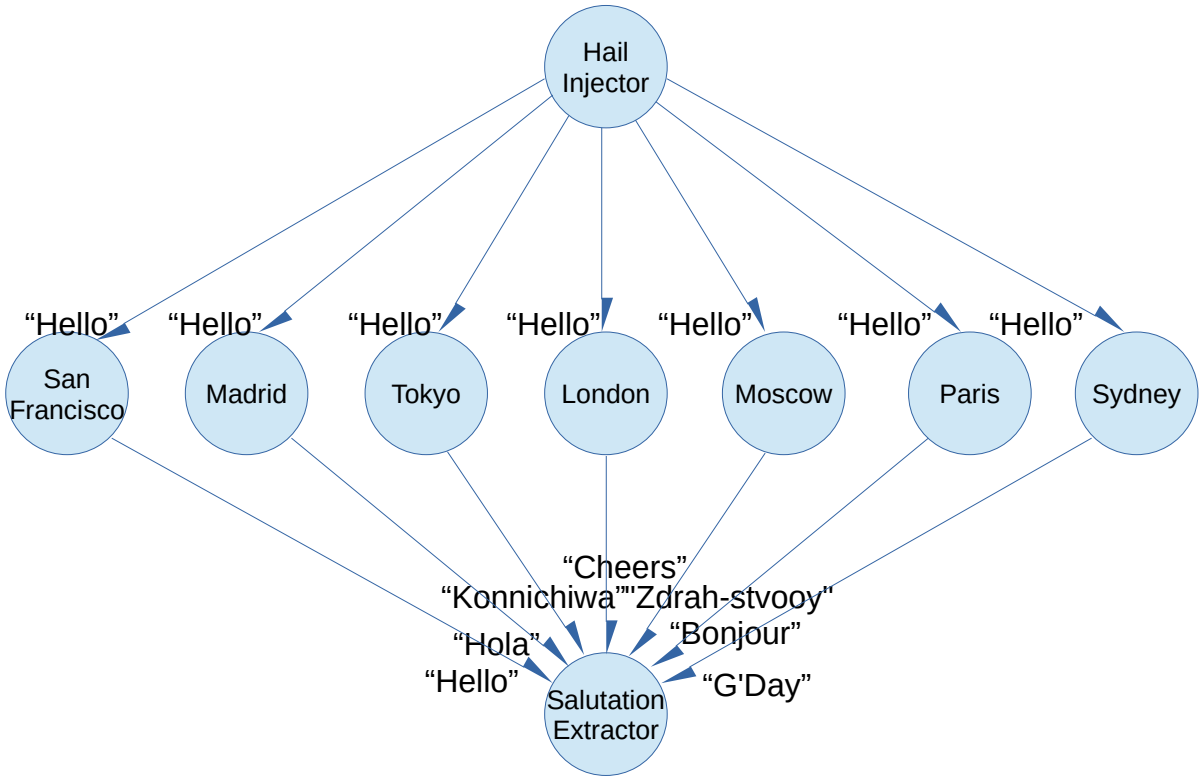


Figure 10: WorldHello Data Flow Model

4.2 WorldHello Cluster Model

The data flow model is useful for defining "what goes where," but it does not tell you "how it got there." The cluster model identifies how each cell will participate in the system. For WorldHello, the most logical cluster model matches the data flow model exactly (see figure 11.) However, we also present alternative models to illustrate the flexibility of the *NeuPaths* framework.

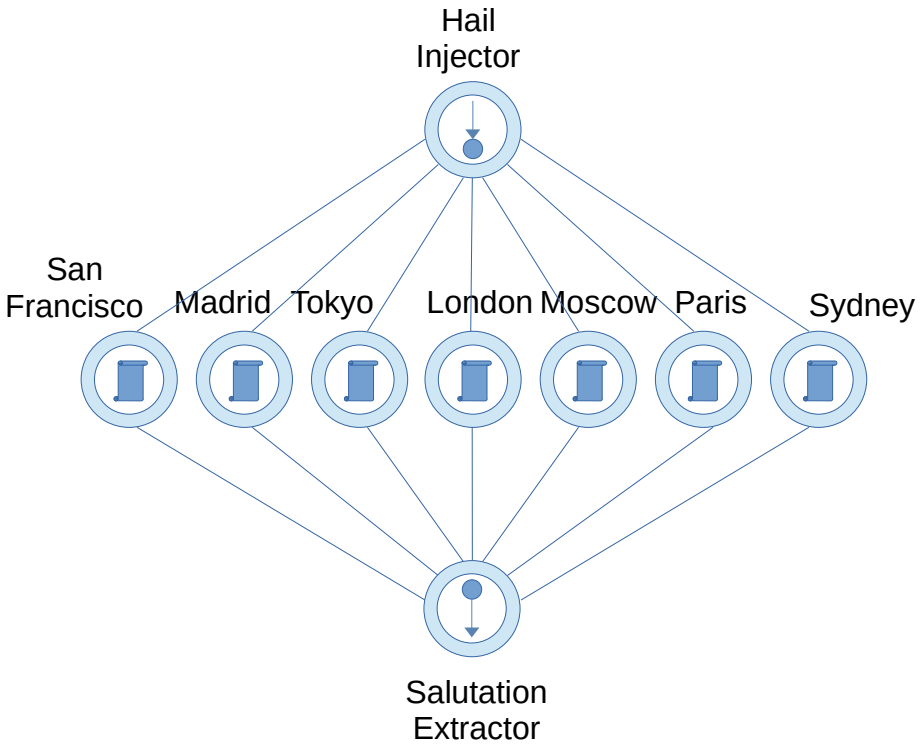


Figure 11: WorldHello Cluster Model

If we imagine that figure 11 represents a broadcast of the hail, then figure 12 might represent a courier physically delivering the hail to each city. In this model, a hail would traverse the entire cluster, and each salutation would take a journey as well.

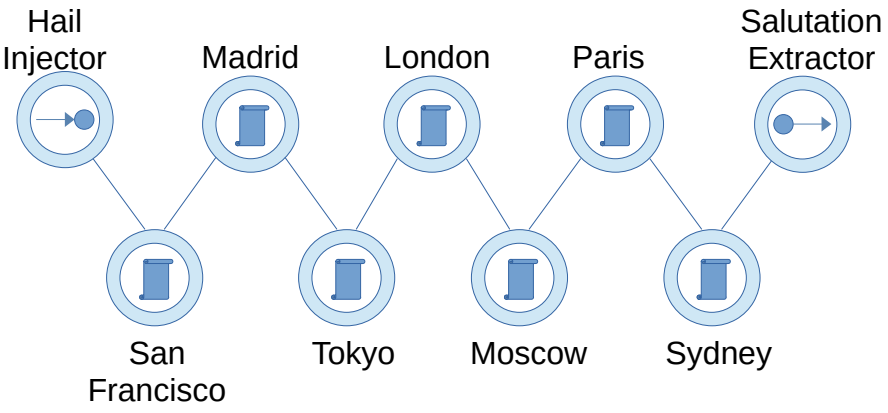


Figure 12: WorldHello Alternate Cluster Model

The next cluster model places the injector at the center of the cluster, where it broadcasts the hail and also routes the salutations to the extractor.

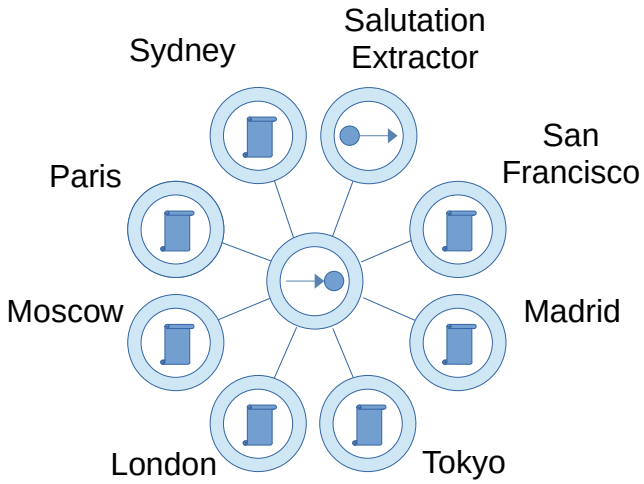


Figure 13: WorldHello Alternate Cluster Model

Yet another cluster model places the extractor at the center of the cluster, where it routes the hail and waits for salutations.

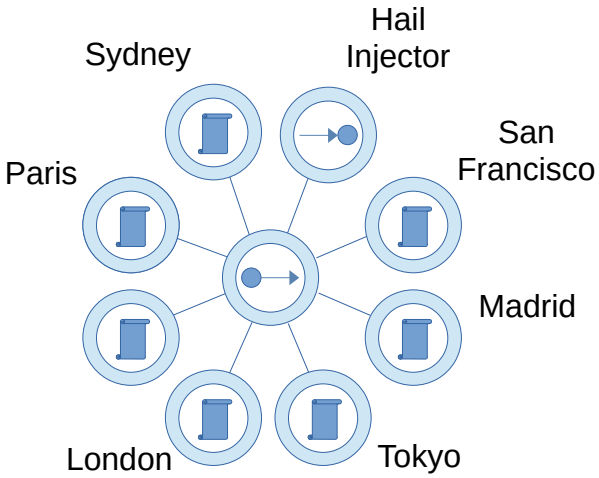


Figure 14: WorldHello Alternate Cluster Model

This last cluster model uses a central router to transport the hail and salutations.

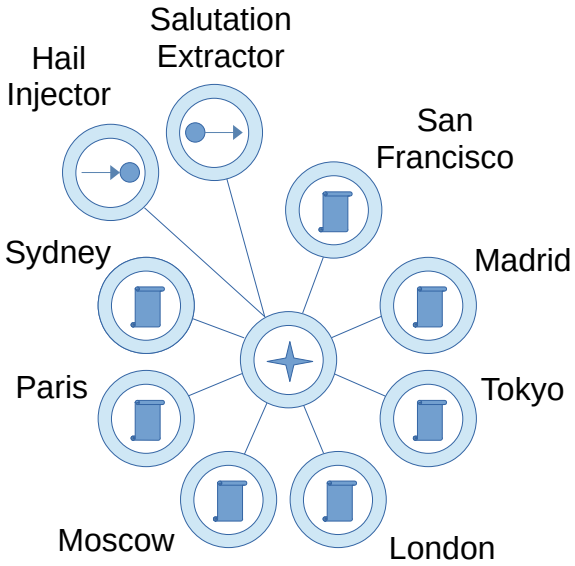


Figure 15: WorldHello Alternate Cluster Model

There are many more possibilities, but regardless of the chosen model, the stimuli and activators do not change.

The cluster model is useful when tuning the cellular system for performance, reliability, scalability, security and resource utilization.

4.3 WorldHello Network Model

The network model indicates where the cellular system lives; it could be on a single host computer, on several host computers in a single intranet, or spread across multiple interconnected data networks. For the WorldHello example, the entire cell cluster will reside on a single host computer:

Host	Cell	Synapse(s)
any host	Hail Injector	Network#Unicast#Listener#Request#30001
	Salutation Extractor	Network#Unicast#Listener#Reply#30002
	London	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002
	Madrid	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002
	Moscow	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002
	Paris	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002
	San Francisco	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002
	Sydney	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002
	Tokyo	Network#Unicast#Peer#Request#30001, Network#Unicast#Peer#Reply#30002

Notice that the system is split into two domains: Request and Reply. While using the global domain would have been simpler, this cluster model will better demonstrate the techniques used for debugging a *NeuPaths* cellular program. Here is the resulting cluster model:

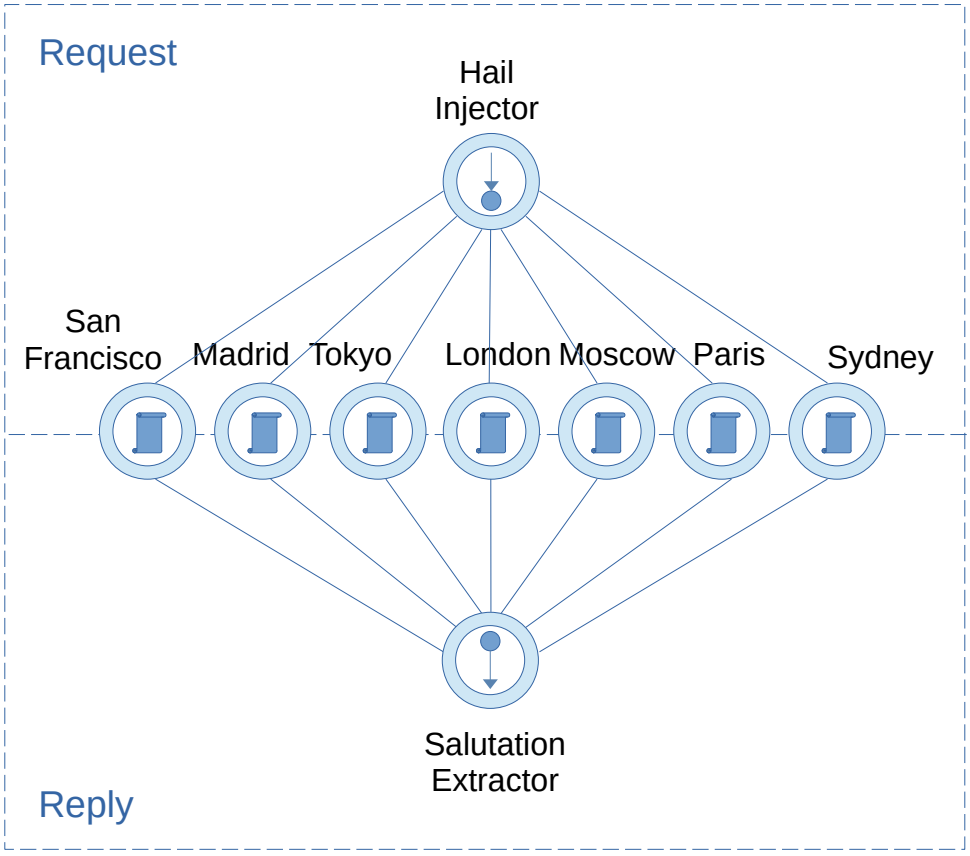


Figure 16: WorldHello Final Cluster Model

The data network model is useful for determining the following:

- Synapse names for all cells.
- Best transport types to use for the problem.
- Network and security issues to be considered.

4.4 Implementation

With the high-level design tasks completed, it is now time to implement WorldHello. The data flow model suggests that two stimulus types are needed: Hail and Salutation. The Hail stimulus does not need to convey any information; receipt alone acts as a signal. We can use the neupaths.stim.DateStimulus type for hailing the cities. The Salutation stimulus will be a custom

stimulus that provides city name, salutation and a timestamp. In considering that each city cell will behave similarly, it is possible to create a single parameterized activator.

4.4.1 WorldHelloSalutation Stimulus Class

The WorldHelloSalutation class extends the neupaths.api.Stimulus class in the *NeuPaths* API. The Stimulus class constructor expects a type name and type ID, both of which should be unique for the class (i.e. apply to all instances of the class). The *NeuPaths* runtime uses these values for type checking of stimuli during execution. The static final TYPE_NAME and TYPE_ID members ensure that the class has a unique identity to the runtime. The *GenerateStimulusType* tool in the neupaths.util package generates a Java source file with a unique TYPE_ID. This tool is helpful for creating a new stimulus type. The WorldHelloSalutation class is a simple container for the city name, salutation and date/time of reply. The *NeuPaths* runtime takes care of all data marshaling for network transport. The serialVersionUID value is generated by the *serialver* Java SDK utility.

WorldHelloSalutation.java

```
import neupaths.api.Stimulus;
import java.util.Date;
import java.util.UUID;

public class WorldHelloSalutation extends Stimulus
{
  public WorldHelloSalutation (String cityName, String salutation)
  {
    super(TYPE_NAME, TYPE_ID);

    this.cityName = cityName;
    this.salutation = salutation;
    date = new Date();
  }

  @Override
  public String toString()
  {
    return "\"" + salutation + "\" from " + cityName + " on " + date;
  }

  public String cityName;
  public String salutation;
  public Date date;

  public static final String TYPE_NAME = "WorldHelloSalutation";
  public static final UUID TYPE_ID =
      UUID.fromString("8c55f6e4-942f-4e3b-a822-2562425c1a94");

  static final long serialVersionUID = -2731912013491025319L;
}
```

4.4.2 WorldHelloActivator Activator Class

The WorldHelloActivator class extends the neupaths.api.Activator abstract class in the *NeuPaths* API. An activator must implement the evaluate method; this is where all stimuli processing takes place. The evaluate method sends a WorldHelloSalutation in reply. The WorldHelloActivator also

overloads the start method, using it to retrieve two properties during initialization: *city_name* and *salutation*. This allows a WorldHelloActivator to parameterize its processing. Refer to the *NeuPaths* API documentation for neupaths.api.Activator class specialization instructions.

An activator requires three things: receptors, transmitters and subscriptions. These values are hardcoded for the WorldHelloActivator because it behaves the same way for all city cells.

WorldHelloActivator.java

```
import neupaths.api.*;
import neupaths.stim.*;

public class WorldHelloActivator extends Activator
{
  public WorldHelloActivator ()
  {
    super("WorldHelloActivator",
          new ReceptorSpec[] {
            new ReceptorSpec("Hail",
                             ReceptorMode.NON_BUFFERED,
                             DateStimulus.TYPE_ID)
          },
          new TransmitterSpec[] {
            new TransmitterSpec("Salutation",
                                WorldHelloSalutation.TYPE_ID,
                                StimulusTrace.ENABLED)
          },
          new LogicSubscriptionSpec[] {
            new LogicSubscriptionSpec("HailInjector",
                                      "Hello",
                                      "Hail",
                                      "Request",
                                      TransactionFilter.DISABLED)});
  }

  @Override
  public void start ()
  {
    cityName = getProperty("city_name");
    salutation = getProperty("salutation”);
  }

  @Override
  protected void evaluate ()
  {
    DateStimulus s = getStimulus("Hail");

    System.out.println("... " + cityName + " was hailed on " + s);

    WorldHelloSalutation response =
        new WorldHelloSalutation(cityName, salutation);

    setStimulus("Salutation", response);
  }

  private String cityName;
  private String salutation;
}
```

4.4.3 Cell and Cluster Definition Files

This implementation will use the neupaths.api.CellCluster class to create and run the WorldHello system. The CellCluster class reads a Cluster Definition XML file to determine which cells to deploy. Each cell in the cluster is defined by a Cell Definition XML file used by the neupaths.api.CellFactory class. Refer to the *NeuPaths* API documentation for details on these classes and the definition file schemas.

For WorldHello, the cluster definition file is quite simple. There are definitions for the injector, the extractor, the seven cities and two event cells.

WorldCluster.xml

```
<cell_cluster>
  <name>WorldCluster</name>
  <cell_definitions>
    <cell_definition>cfg/HailInjector.xml</cell_definition>
    <cell_definition>cfg/SalutationExtractor.xml</cell_definition>
    <cell_definition>cfg/London.xml</cell_definition>
    <cell_definition>cfg/Madrid.xml</cell_definition>
    <cell_definition>cfg/Moscow.xml</cell_definition>
    <cell_definition>cfg/Paris.xml</cell_definition>
    <cell_definition>cfg/SanFrancisco.xml</cell_definition>
    <cell_definition>cfg/Sydney.xml</cell_definition>
    <cell_definition>cfg/Tokyo.xml</cell_definition>
    <cell_definition>cfg/RequestLog.xml</cell_definition>
    <cell_definition>cfg/ReplyLog.xml</cell_definition>
  </cell_definitions>
  <trace_logging_enabled>true</trace_logging_enabled>
</cell_cluster>
```

The injector's class definition file lists a single Listener synapse to which all of the city cells connect. It also defines the injector's transmitter.

HailInjector.xml

```
<injector_cell>
  <name>HailInjector</name>
  <synapses>
    <synapse>Network#Unicast#Listener#Request#30001</synapse>
  </synapses>
  <transmitter>
    <name>Hello</name>
    <stimulus_class>neupaths.stim.DateStimulus</stimulus_class>
    <trace>enabled</trace>
  </transmitter>
</injector_cell>
```

The extractor's cell definition file also lists a single Listener synapse to which all of the city cells connect. It specifies the subscription for the salutations. The subscription uses a regular expression for the producer cell name that matches all of the city names.

SalutationExtractor.xml

```
<extractor_cell>
  <name>SalutationExtractor</name>
  <synapses>
    <synapse>Network#Unicast#Listener#Reply#30002</synapse>
  </synapses>
  <subscription>
    <cell_name>.*_City</cell_name>
    <transmitter_name>Salutation</transmitter_name>
    <domain>Reply</domain>
  </subscription>
</extractor_cell>
```

The city cell definition files resemble each other very closely. All have two synapses: a Peer for connecting to the injector, and another Peer for connecting to the extractor. Each city cell uses two properties to differentiate itself: *city_name* and *salutation*. The London cell definition file shows the definition format.

London.xml

```
<logic_cell>
  <name>London_City</name>
  <synapses>
    <synapse>Network#Unicast#Peer#Request#30001</synapse>
    <synapse>Network#Unicast#Peer#Reply#30002</synapse>
  </synapses>
  <activators>
    <activator>
      <class>WorldHelloActivator</class>
      <properties>
        <property>
          <name>city_name</name>
          <value>London</value>
        </property>
        <property>
          <name>salutation</name>
          <value>Cheers</value>
        </property>
      </properties>
    </activator>
  </activators>
</logic_cell>
```

Finally, there is an event cell for each of the domains:

RequestLog.xml

```
<event_cell>
  <name>RequestDomainLogger</name>
  <synapse>Network#Unicast#Peer#Request#30001</synapse>
  <output_file>worldhello_request_events.out</output_file>
</event_cell>
```

ReplyLog.xml

```
<event_cell>
  <name>ReplyDomainLogger</name>
  <synapse>Network#Unicast#Peer#Reply#30002</synapse>
  <output_file>worldhello_reply_events.out</output_file>
</event_cell>
```

4.4.4 Main Class

The Main class implements the Java main method, which instantiates and starts the cell cluster. After injecting the Hello stimulus, the main method extracts each Salutation stimulus. In this contrived example, the exact number of expected responses is known, so the cell network is shut down when all replies have been received.

Main.java

```
import neupaths.api.*;
import neupaths.stim.*;

import java.util.LinkedList;

public class Main
{
  public static void main (String[] args)
  {
    try
    {
      System.out.println("Initializing Cell Network...");

      // Construct and start the cluster
      CellCluster cluster =
          new CellCluster("cfg/WorldCluster.xml");
      cluster.start();

      // Retrieve injector and extractor from cluster
      InjectorCell hail = cluster.getCell("HailInjector");
      ExtractorCell salutations = cluster.getCell("SalutationExtractor");

      // Give cluster time to initialize
      try
      {
        Thread.sleep(2000);
      }
      catch (InterruptedException ie)
      {
        /* ignore */
      }

      // Hail the cities
      System.out.println("Hailing the cities...");
      hail.inject(new DateStimulus());

      // Display salutations
      int responseCount = 0;
      while (responseCount < 7)
      {
        WorldHelloSalutation salutation = salutations.extract();
        responseCount++;
```

```
        System.out.println("--> " + salutation);
      }

      System.out.println("All cities have reported.  Terminating...");

      // Give cluster time to process final events
      try
      {
        Thread.sleep(1000);
      }
      catch (InterruptedException ie)
      {
        /* ignore */
      }

      // Stop all cells
      cluster.stop();

      System.out.println("Cell Network shutdown.  Good bye.");
    }
    catch (NeuPathsException npe)
    {
      System.out.println("Fatal error occurred: " + npe);
    }
  }
}
```

4.4.5 Output

```
Initializing Cell Network...
Hailing the cities...
... Madrid was hailed on Sun Jan 26 12:50:18 EST 2025
... Tokyo was hailed on Sun Jan 26 12:50:18 EST 2025
... Sydney was hailed on Sun Jan 26 12:50:18 EST 2025
... Moscow was hailed on Sun Jan 26 12:50:18 EST 2025
... San Francisco was hailed on Sun Jan 26 12:50:18 EST 2025
... Paris was hailed on Sun Jan 26 12:50:18 EST 2025
... London was hailed on Sun Jan 26 12:50:18 EST 2025
--> "Konnichiwa" from Tokyo on Sun Jan 26 12:50:18 EST 2025
--> "Bonjour" from Paris on Sun Jan 26 12:50:18 EST 2025
--> "Zdrah-stvooy" from Moscow on Sun Jan 26 12:50:18 EST 2025
--> "Hola" from Madrid on Sun Jan 26 12:50:18 EST 2025
--> "G'Day" from Sydney on Sun Jan 26 12:50:18 EST 2025
--> "Hello" from San Francisco on Sun Jan 26 12:50:18 EST 2025
--> "Cheers" from London on Sun Jan 26 12:50:18 EST 2025
All cities have reported.  Terminating...
Cell Network shutdown.  Good bye.
```

5 Debugging

A cellular system's concurrent, asynchronous and possibly non-deterministic processing can pose a significant challenge when debugging. To assist with debugging, the *NeuPaths* framework includes logging facilities.

The neupaths.api.Activator class provides logEvent methods that can be used in the start, evaluate and stop methods. The logEvent methods can be used in mainline code and will only produce event

stimuli when the corresponding event facilities are enabled. User-level logging is enabled by default. This encompasses logging of the AUDIT, INFORMATION, WARNING and ERROR event types. The *NeuPaths* runtime may also log INFORMATION, WARNING and ERROR event types. The RUNTIME event type is reserved by *NeuPaths* to assist with product support. Runtime logging is disabled by default. Trace and Debug logging are also disabled by default. The TRACE and DEBUG event types will prove indispensable for *NeuPaths* API clients. The *NeuPaths* runtime will provide some trace information, such as stimulus trace on ingress and egress from a cell and subscription registration information. Users can add their own TRACE events to assist with debugging. Users are also encouraged to use DEBUG events. The logging facilities can be controlled for each cell individually or for a whole cluster, either within Cluster/Cell definition files or programmatically via the API.

Developers can include event cells (neupaths.api.EventCell) directly in their clusters or use the neupaths.util.EventLogger tool. Developers can also make their own specialization of the neupaths.api.EventActivator class and use it with the EventCell class. In cases where a cell is having issues joining a cluster, the cell's enableDebutOutputLogging method can be used to direct events to the processes' standard output. Refer to the *NeuPaths* API documentation for more details.

Here are some helpful practices for debugging:

- Use an EventCell in each domain in the system.
- Turn on Trace and Debug logging for all cells.
- Set the subscription trace interval. The subscription refresh interval is 1500 milliseconds by default, so a 2000 millisecond subscription trace interval would be sufficient.
- Add TRACE and DEBUG events to assist with debugging.
- Analyze stimuli paths using ingress and egress trace logs.
- Analyze subscription registrations for each cell.

The stimuli ingress and egress events can be used to trace stimuli through the system. The following events produced by WorldHello trace the Hello and Salutation stimuli for the London cell.

```
Type:   TRACE
Time:   01/27/25 21:02:15.493000000
Source: HailInjector (e38e1f99-fdd5-48a2-ba88-1331214a0e33) EGRESS
Details:
Stimulus trace for type 'DateStimulus' (0c00c56e-2d00-4581-8ed9-04012d157a05) on
synapse a8cf77b0-dcca-45a1-b314-65d004ab419a: HailInjector

Type:   TRACE
Time:   01/27/25 21:02:15.502000000
Source: London_City (4b057fa4-9db4-4c34-901a-81cfd3f343d4) INGRESS
Details:
Stimulus trace for type 'DateStimulus' (0c00c56e-2d00-4581-8ed9-04012d157a05) on
transmitter 'Hello': HailInjector => London_City
```

```
Type:   TRACE
Time:   01/27/25 21:02:15.542000000
Source: London_City (4b057fa4-9db4-4c34-901a-81cfd3f343d4) EGRESS
Details:
Stimulus trace for type 'WorldHelloSalutation' (279ed6b5-0c46-4174-99c6-
af86f80f3835) on synapse 8d6ae73a-8e3f-4c70-86e9-501ae12b54ea: London_City

Type:   TRACE
Time:   01/27/25 21:02:15.557000000
Source: SalutationExtractor (94817e08-a3fb-4cf9-8776-e3e5fe98192a) INGRESS
Details:
Stimulus trace for type 'WorldHelloSalutation' (279ed6b5-0c46-4174-99c6-
af86f80f3835) on transmitter 'Salutation': London_City => SalutationExtractor
```

Subscription trace events can be used to trace subscription registrations through the system. The following are subscription trace events from WorldHello. Note that only London is included for the city cells. The remaining city cells will have the same registrations. The HailInjector cell only has one registered subscription. That subscription will note all of the synapses interested in the Hello stimulus, namely each synapse with a city cell. Each city cell has only one registered subscription for its Salutation stimulus. This subscription is advertised by the SalutationExtractor cell. Notice that the SalutationExtractor cell does not have any subscription registrations; the extractor is the only consumer in the Reply domain, and therefore has not received any registrations.

```
Type:   TRACE
Time:   01/27/25 21:02:15.422000000
Source: HailInjector (e38e1f99-fdd5-48a2-ba88-1331214a0e33) Nucleus Subscription
Trace
Details:
Registered Subscriptions:
  Cell: HailInjector, Transmitter: Hello, Domain: Request

Type:   TRACE
Time:   01/27/25 21:02:15.426000000
Source: London_City (4b057fa4-9db4-4c34-901a-81cfd3f343d4) Nucleus Subscription
Trace
Details:
Registered Subscriptions:
  Cell: .*_City, Transmitter: Salutation, Domain: Reply

Type:   TRACE
Time:   01/27/25 21:02:15.425000000
Source: SalutationExtractor (94817e08-a3fb-4cf9-8776-e3e5fe98192a) Nucleus
Subscription Trace
Details:
Registered Subscriptions:
```

6 Performance and Resources

A *NeuPaths* system can consume a lot of resources. Each cell consists of several threads, each synapse consumes network resources, and receptors and transmitters use heap memory.

The cluster model can have a large impact on resource utilization. Subscribing to stimuli from distant cells uses more resources than from a nearby cell (in terms of network and processor utilization.) A system with a complex model that includes cycles will rely heavily on duplicate stimuli detection, which requires resources to manage (in terms of heap and processor utilization.)

Use of domains can influence system performance. Subscriptions periodically propagate throughout a system, which can add significantly to system traffic. Subscriptions in the global domain will reach every cell in a system. For large systems, this can consume important resources. Subscriptions in a particular domain will only reach cells participating in the domain, and since stimuli only go where subscriptions have gone, the stimuli will also remain within the domain.

The tuning of performance and resource utilization is an iterative process. The cluster model can be refined, and in most cases refinement will not change the data flow model. The subscription refresh and duplicate detection intervals can be tuned for each cell independently. The number and location of JVM instances can be refined to improve processor and memory utilization.

7 Security

All communication in a *NeuPaths* system is encrypted using Blowfish, a public domain encryption algorithm. Users have the option of encrypting stimuli with their own secret key as well. The *GenerateCryptoKey* tool in the neupaths.util package will generate a unique secret key, which can be provided to the cell classes during construction or in Cluster/Cell definition files. Refer to the *NeuPaths* API documentation for details.

8 Hints and Tips

- A cell system at initialization is similar to a human brain waking from slumber. It can take a few seconds for the system to get its bearings. Wait a few subscription refresh cycles before injecting stimuli.
- Use of domains will improve performance and prevent flooding the cluster. Avoid using the global domain in complex clusters.
- Use regular expressions in subscriptions, wherever possible, to reduce the burden of subscription management in the cluster.
- Partition a large system into multiple JVM executions to parallelize cluster creation. The *NeuPaths* framework is heavily multi-threaded, which can impact performance of the JVM during initialization.

9 Glossary

Activator	Algorithm that consumes and/or transforms stimuli.
Cell	Autonomous agent that processes stimuli.
Cluster	A group of interconnected cells. A sub-graph of a cellular system.
Domain	A logical association of communication paths within a system. Subscriptions are restricted to their designated domains, which in turn restricts stimuli transmission. The *NeuPaths* global domain ("@") is always unrestricted.
Nucleus	Routes the stimuli to interested activators and neighboring cells.
Receptor	A named, type-safe receptacle for stimuli.
Stimulus	Atomic data element that is produced and consumed by cells.
Subscription	The mechanism a cell uses to express interest in a stimulus. A subscription consists of the tuple (*ProducerCellName*, *ProducerTransmitterName*, *ReceptorName*, *Domain*)
Synapse	Transports stimuli between cells. An abstraction for underlying network technologies.
System	An asynchronous, distributed data processor composed of one or more clusters.
Transmitter	A named, type-safe emitter of stimuli.