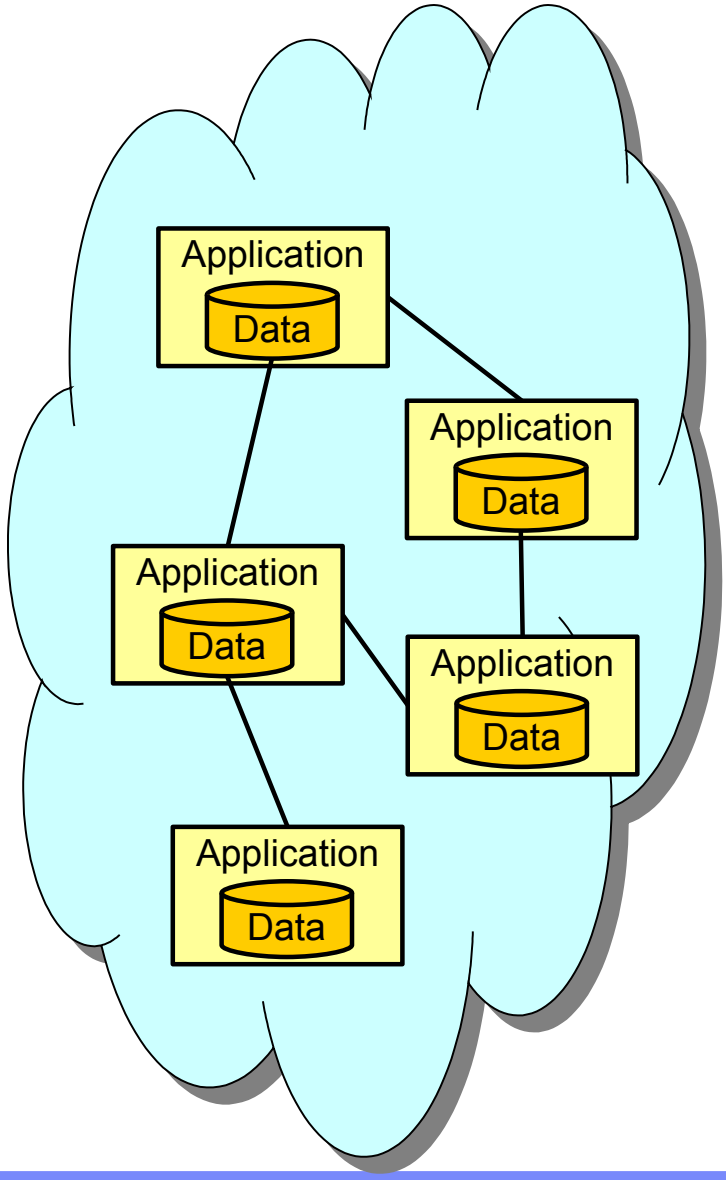
IBM Research

Distributed State Machine Rule Engine Tutorial

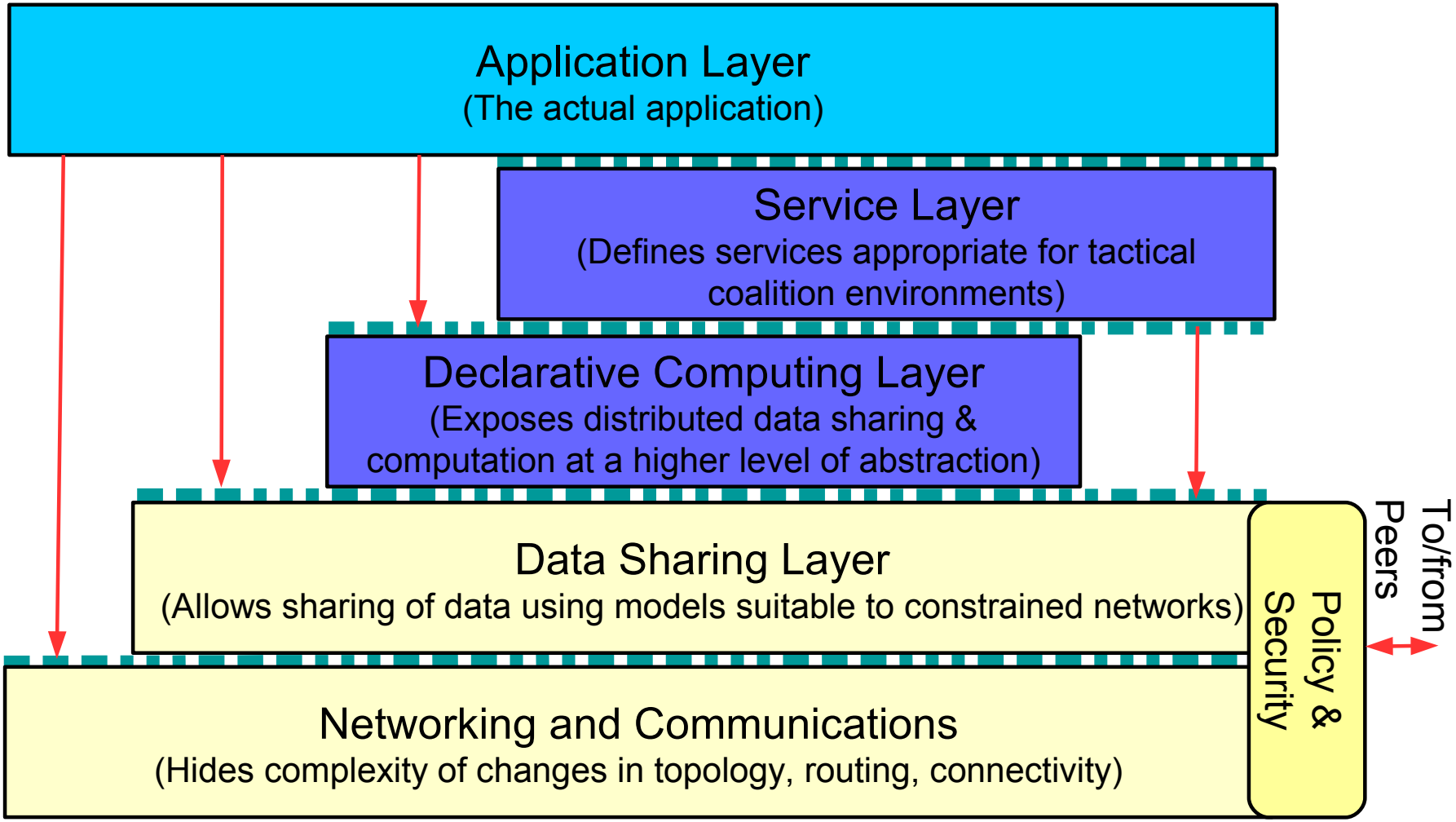
David Wood.
IBM T. J. Watson Research Center

Distributed State Machine Apps

- **Applications collaborate to solve a distributed problem.**
  - Network management
  - Sensor data collection and processing
- **Data collected locally and shared with others as defined by the application.**
  - Both relational and Java data models supported
- **Topology-based continuous or synchronous queries of shared data**
  - Interactions by topological relationships
- **Policy enforcement over data sharing**
  - Filtering and access control on both inbound and outbound data and queries.



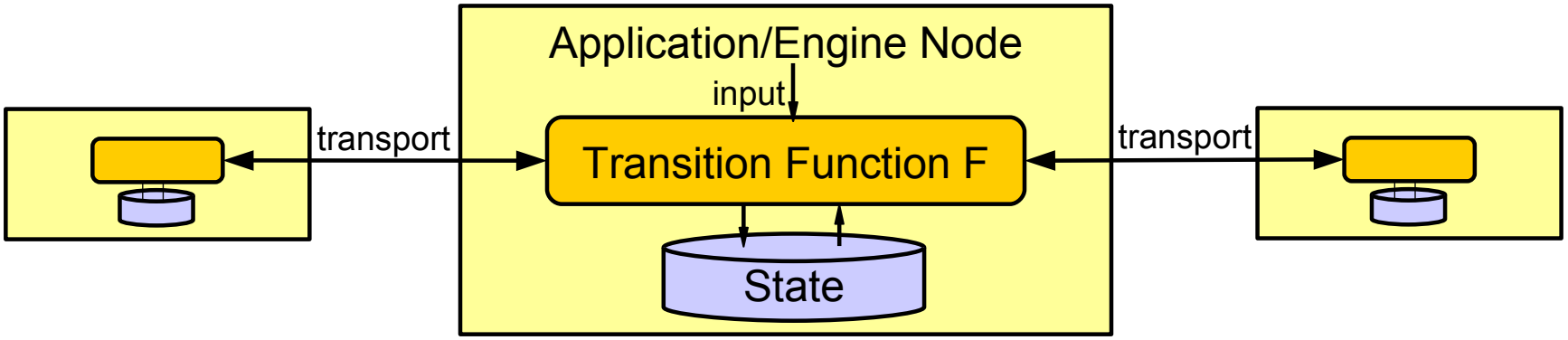
Layered Distributed Application Framework



And a suite of development tools to make it easier to use

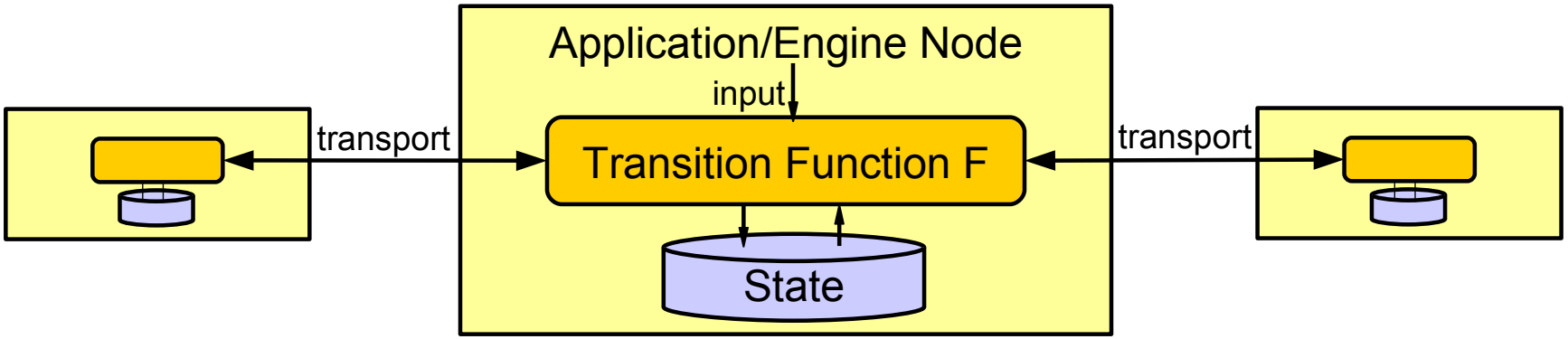
Distributed State Machine Rule Engine

- **Rule language provides ability to...**
  - Define state schema and data model
  - Define state transition function, F, to manipulate local relational tables (aka state)
    - Next State = F(Current State, Input/Transport State)
  - Send/receive state to/from other engines
- **Rules are evaluated in response to the availability of new state**
  - New state comes in the form of input or transport rows
- **State stored as relational data**



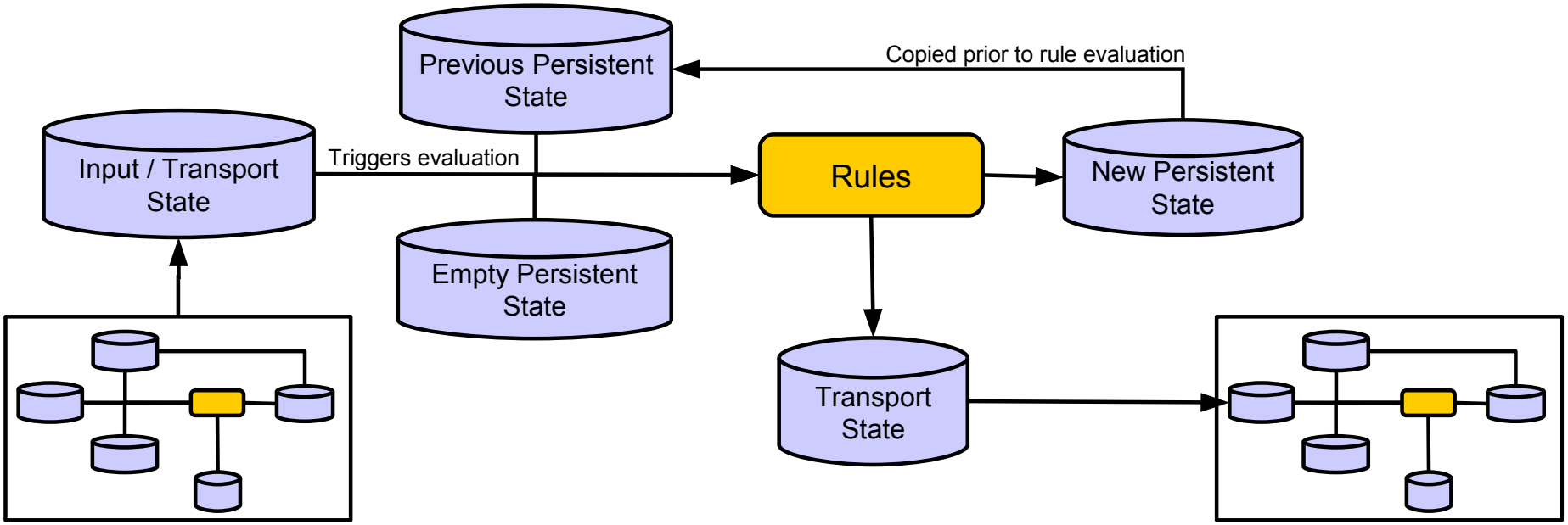
DSM Rule Language – A Quick Example

```
// receives input from local application running rules engine.
input send_message(int msgid, char[128] dest, char[128] msg);

// Defines table holding in/outbound messages
transport message(int msgid, char[128] msg);

// Sends message as defined by input received locally.
message(msgid, msg)@dest if send_message(msgid, dest, msg);

// Sends response to messages received from other engines, unless it is an ACK.
message(0, “ACK:” + msgid)@src if message(msgid, *)@src : msgid != 0;
```

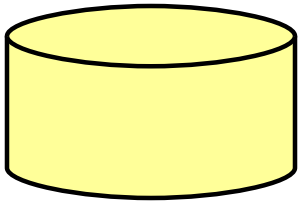


State Transition Detail

- **Input/Transport tuples applied to engine trigger new evaluations**
- **Previous state and input/transport tuples used to create new state**
- **State manipulated through rule statements**
  - **A(b) if B(b)** – copies all values from the current state in B to the current state in A
  - **C(b) if prev A(b)** – copies values in A from previous state into C of the next state.



DSM State



- **State - One or more relational tables**
  - Schema and contents of state tables are application defined
    - Column types: int, float, double, char[], timestamp
- **Tables are typed**
  - *system* – contains engine-provided run-time data, read-only.
  - *input* – new local state to be applied to existing state, read-only.
  - *transport* – used to send/receive state to/from other engines, read-write.
  - *persistent* – application state held across rule evaluations, read-write.
  - *action* – define Java methods called after rule evaluations, read-write.

State Table Declarations

- **Tables are defined with a type, name and column types and names.**
  - <type> <name> (<column list>);
- **Examples**
  - input my_observations(char[128] node, double opinion);
  - persistent my_opinions(char[128] node, double opinion);
  - persistent individual_observations(char[128] source, char[128] node, double opinion);
  - transport shared_observations(char[128] node, double opinion);
  - action my_actions(int isSync, char[128] methodName, int x, double y);
- **System tables**
  - system peers(char[128] node, char[64] relationship);
    - Holds all topology relationships for all known nodes – neighbor, parent, reachable, etc.
  - system peer_change(char[128] node, char[64] status);
    - Holds *changes* in peer polulation for a given node. Status is either 'added' or 'removed'.

State Change Rules

- **General format**
  - <assigned tuple> '**if**' <tuple references> ('**:**' <conditions>)? ('**:**' '[' <modifier list> ']')? '**;**'
    - These are assertions about new states that are made true if the state implied on the RHS of the ***if*** exists.
  - <assigned tuple> - defines table and values to be assigned in table
  - <tuple references> - lists 1 or more tuples whose values can be applied to the new state. Joins between tuples can be implied when more than 1 tuple is used.
  - <conditions> - optional logical expressions on elements of the RHS tuples.
  - <modifiers> - optional rule evaluation controls.
- **Example: A(x,y) if B(x,z), C(y,z) : z > 10 : [MSEC=2000];**
  - A(x,y) if – appends to A the values of x and y, defined by the RHS.
  - B(x,z),C(y,z) – selects values of x and y from rows where z is the same in B and C.
  - z > 10 – requires that the value of z from B and C is larger than ten.
  - [MSEC=2000] – causes this rule to be considered for evaluation only every 2 seconds.

State Change Rules (cont)

- **Current and previous persistent state**
  - Current persistent state is always empty at the beginning of rule evaluation
  - Previous state is accessed using the ***prev*** keyword on persistent tables.
    - **A(y) if prev A(y);** // Copies previous state of A to current state.
- **RHS Tuple references and joins**
  - Table joins
    - **A(y) if B(5,y);** // Copies values of y from B where the first column has a value of 5
    - **A(x,y) if B(a,x), C(a,y);** // Copies values of B.x and C.y where the 1st columns are equal
  - Negative table joins
    - **A(x) if B(a,x), not prev C(a);** // Copy values from B.x where the corresponding B.a is not in previous C
- **Logical expressions on combinations of row/column values**
  - **A(y) if B(x,y) : x > 5;** // Copies values of y from B where the 1st column has a value larger than 5
  - **A(x,y) if B(a,x), C(b,y) : a > b;** // Copies values of B.x and C.y where the B.a > C.b
- **Rule evaluation modifiers**
  - **A(y) if B(5,y) : [msec=2000];** // Don't evaluate this more often than every 2 seconds.
  - **A(y) if B(5,y) : [choose(3,2)];** // Produces maximum of 3 rows and a minimum of 2, or none.

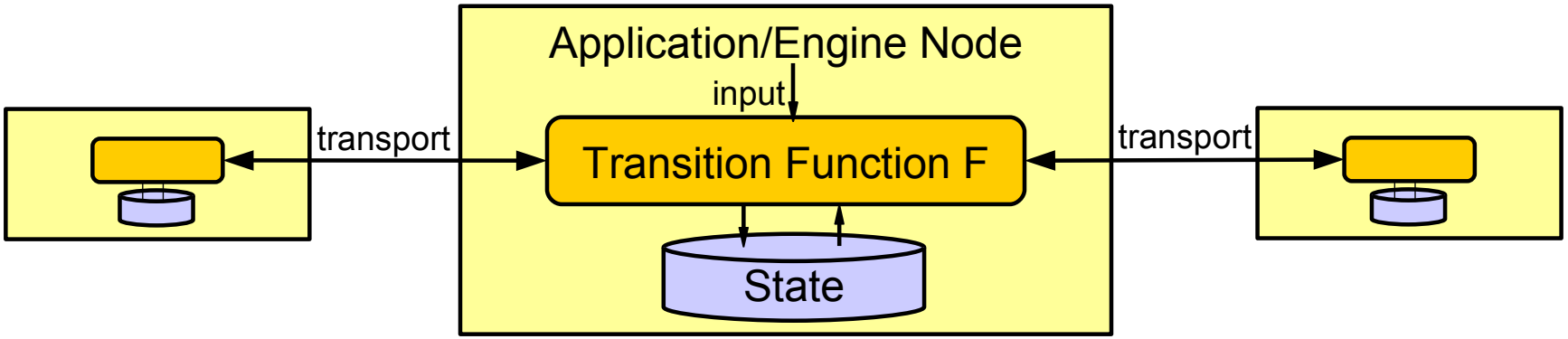
Ahhh Transport Tuples!

- **Transport tuples are sent/received to/from other engines**
- **Engine instance identifiers are used to address tuples**
  - Received tuples include the engine instance id from which they are received
  - Sent tuples include the engine instance id to which they are sent
- **The same notation is used to reference sender/receiver - *table@engineID***
  - engineID is the instance id (i.e. address) of another rule engine
    - destination if on the LHS of the 'if' (i.e., x(y)@z if ...)
    - sender if on the RHS of the 'if' (i.e. … if x(y)@z …)
- **Example**

```
…
transport msg(char[128] text);
persistent log(char[128] source, char[128] msg);
...
msg(“Hello 2U2”)@sender if msg(“Hello There”)@sender;
...
log(sender, text) if msg(text)@sender;
...
```

Returning to Our Quick Example

```
// receives input from local application running rules engine.
input send_message(int msgid, char[128] dest, char[128] msg);

// Defines table holding in/outbound messages
transport message(int msgid, char[128] msg);

// Sends message as defined by input received locally.
message(msgid, msg)@dest if send_message(msgid, dest, msg);

// Sends response to messages received from other engines, unless it is an ACK.
message(0, “ACK:” + msgid)@src if message(msgid, *)@src : msgid != 0;
```



Action Tuples

- **Enables the rules to call Java methods after a rule evaluation**
  - Tables are populated during a rule evaluation.
  - Then read to call methods and parameters defined in the tables.
  - Tables are automatically cleared after actions have been called at the end of an evaluation.

- **A rule table of type 'action' is used to define Java methods.**
  - `action my_actions(int sync, char[128] methodName (, additional columns)? );`
  - The action table must have columns 1 and 2 as above.
  - Additional columns can hold values that are passed to the Java method.
  - Any number of action tables may be defined.

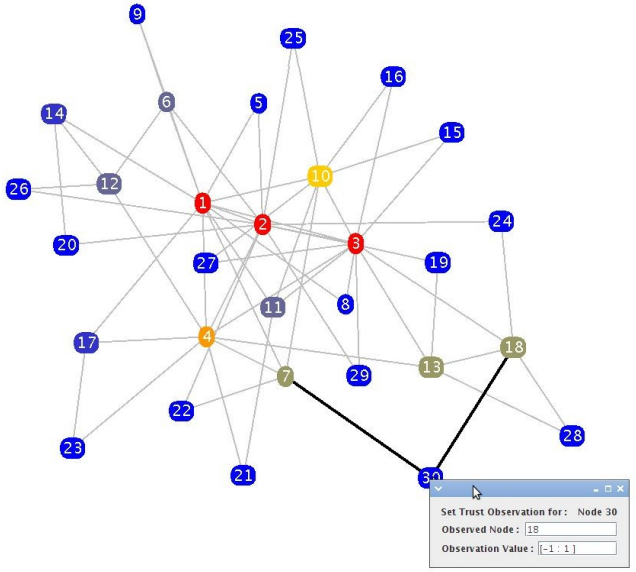
- **Java method must have the following signature:**
  - `public static void MyAction(IRuleEngine r, TupleState begin, TupleState end,...);`
  - `begin/end` are the sets of tuples before and after the current rule evaluations.
  - Values from columns 3..N, if present, are passed to the method as Java objects (not primitive types).
    - In this case, the method signature must expect those objects.

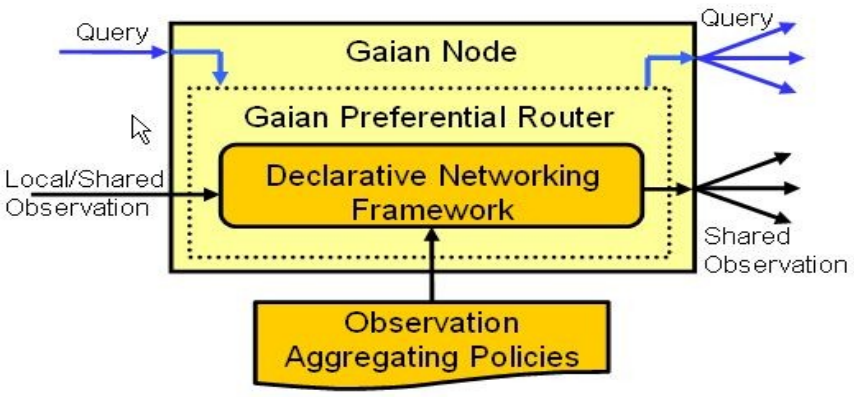
Miscellaneous

- **Rule blocks**
  - ***Block*** keyword may be used to separate rules into evaluation sections.
  - Blocks are evaluated sequentially with recursive evaluation.
- **Functions are provide to operate on values**
  - String – LENGTH, LOCATE, SUBSTR
  - Time – YEAR, MONTH, DAY, HOUR, MIN
  - Aggregation – AVG, MIN, MAX, SUM, COUNT
    - **A(avg(x)) if B(5,x);** // A gets a row with the average of B.x where B's 1st column has value 5.
- **Wildcards**
  - * may be used on the RHS when a column value is to be ignored.
    - **A(x,y) if B(x,y,*), C(x,*);**

Example: Preferential Routing in GaianDB

- **In hostile environments, adapting to compromised network elements is desirable.**
  - A GaianDB query floods the network to aggregate data.
  - All nodes no longer considered equal and viable.
- **Collaboratively monitor and share status of nearby nodes to secure data flows.**
  - e.g. 1 node observes take-over of another
  - Make observations and share with others
  - Negatively observed node routed around/avoided
- **Declarative policies define protocol for sharing and incorporating shared observations.**
  - Propagation lengths
  - Trust levels required before acceptance
  - Aggregation of observations
- **Declarative policy/rule framework**
  - Encapsulates distributed computing model
  - Enables analysis & pre-verification
  - Provides dynamic adaptability, flexibility, brevity





Preferential Routing State Definition

Observations are collected locally and from neighbors, then combined to produce opinions for each node.

- **Persistent tables to hold local opinions and all observations**
  - `persistent my_opinions(char[128] node, double opinion);`
  - `persistent individual_observations(char[128] source, char[128] node, double opinion);`

my_opinions	
node	opinion

individual_observations		
source	node	opinion

- **Input tables to send in local observations**
  - `input my_observations(char[128] node, double opinion);`

my_observations	
node	opinion

- **Used to send/receive observations to/from other Gaian/DSM nodes.**
  - `transport shared_observations(char[128] node, double observation);`

shared_observations	
node	observation

- **System table to get list of neighboring Gaian/DSM nodes.**
  - `system peers(char[128] node, char[64] relationship);`

peers	
node	relationship

Preferential Routing Rules

- **Collect my local, shared and previous existing observations into a single list**
  - `// My locally determined observations`
    `individual_observations("",n,o) if my_observations(n,o);`
  - `// Add in remote observations.`
    `individual_observations(src,n,o) if shared_observations(n,o)@src;`
  - `// Add in previous observations if we don't have a new one`
    `individual_observations(src,n,o) if prev individual_observations(src,n,o),`
    `not individual_observations(src,n,*)`
- **Defines a block boundary before computing on individual_observations**
  - `block;`
- **New opinions are the average of all observations of a node**
  - `my_opinions(n, avg(o)) if individual_observations(*, n, o);`
- **Send local observations to neighbor nodes**
  - `shared_observations(n,o)@neighbor if my_observations(n,o),`
    `peers(neighbor, "Neighbor");`

Command Line Tools

- ***dsmparse <rules file>* -** parses a file, checking for errors, outputting SQL-based rules.
  - Example: ***dsmparse opinions.dsmr***
- ***dsmengine <options> <rules file>*** – starts an engine
  - *-namespace <text>* - defines names space for the engines. All engines using the same name space will be able to exchange tuples.
  - *-instance <text>* - defines the instance id for the engine. Optional, but should be set if using ***tuple*** command line tool. Due to a current bug, instance ids should not be integers in order to work properly with the ***tuple*** tool.
  - *-interactive* – causes the executable to wait for user input before exiting, otherwise engine must be killed externally (e.g., linux kill).
  - Example: ***dsmengine -namespace opinions -instance foo opinions.dsmr***
- ***tuple <command> <namespace> <app id> <table> [<tuples>|<csv file>]–*** inserts/reads to/from an engine.
  - <command> - *insert* or *read* – indicates the addition or reading of tuples to/from the indicated engine.
  - <namespace> - name space used by the application/engine.
  - <app id> - id of a DSM application/engine. Probably a value given to dsmengine's -instance option.
  - <tuples> - specifies the name of a state table and the values for the columns for insert command.
  - <csv file> - a file of comm-separate tuple values, first line is the names of columns of the indicated table.
  - Example: ***tuple insert opinions foo my_observations node=b opinion=.3***
  - Example: ***tuple insert opinions foo my_observations observations.csv***
  - Example: ***tuple read opinions foo my_opinions***

Running the Opinion Rules Under Simulation

see $DSM_HOME/samples/com/ibm/watson/dsm/samples/opinions/opinions.sh

```
1)  rules=$DSM_HOME/samples/com/ibm/watson/dsm/samples/opinions/opinions.dsmr

2)  # Create 3 engines with instances a,b,c each using the same rules file
3)  for i in a b c; do
4)     dsmengine -namespace opinions -instance $i $rules > engine-$i.out 2>&1 &
5)  done
6)  …

7)  # Send some opinions into each of the engines.
8)  tuple insert opinions a my_observations node=b opinion=.3   # a observes b with opinion .3
9)  ...
10)tuple insert opinions b my_observations node=a opinion=.6   # b observes a with opinion .6
11)...
12)tuple insert opinions c my_observations node=a opinion=.8   # c observes a with opinion .8
13)…

14)# Kill the engines
15)pids=`ps -elf | grep DSMEngine | grep -v grep | awk '{print $4}'`
16)kill -9 $pids
```

Rule Debugging

- **“A horse, a horse, my kingdom for a horse!” - alas, we do not have a debugger.**
- **Logging is currently the best option.**
  - Add -Dcom.ibm.watson.platform.engine.level=FINE(R|ST)? to dsmengine command.
    - dsmengine -Dcom.ibm.watson.platform.engine.level=FINER ...
- **Looking at engine-c.out from our script**

Engine opinions/c started.

Jun 14, 2013 2:15:09 PM com.ibm.watson.dsm.engine.AbstractRuleEngine.evaluateTupleRules(AbstractRuleEngine.java:461)

FINE: **Running rules over tuples**

Jun 14, 2013 2:15:09 PM com.ibm.watson.dsm.engine.AbstractRuleEngine.evaluateTupleRules(AbstractRuleEngine.java:471)

FINE: Appending tuples TupleSet:TupleSetDescriptor: appDesc=opinions/a, name=**shared_observations**, [ColumnDescriptor: name=**DEST__INTERNAL**, type=String, ColumnDescriptor: name=**SRC__INTERNAL**, type=String, ColumnDescriptor: name=NODE, type=String, ColumnDescriptor: name=OPINION, type=Double, ColumnDescriptor: name=TOC__INTERNAL, type=Timestamp][

Tuple[TupleEntry: **value=c**, TupleEntry: **value=a**, TupleEntry: value=b, TupleEntry: value=0.3, TupleEntry: value=1970-01-01 14:15:09.026]

]

…

FINE: **Completed evaluation over tuples. Sending transport tuples..**

...

FINE: **After complete evaluation...Application opinions/c contains the following tables...**

…

Table: my_opinions, TupleSet:TupleSetDescriptor: appDesc=opinions/c, name=**my_opinions**, [ColumnDescriptor: name=NODE, type=String, ColumnDescriptor: name=OPINION, type=Double, ColumnDescriptor: name=TOC__INTERNAL, type=Timestamp][

Tuple[TupleEntry: value=a, TupleEntry: value=0.7, TupleEntry: value=1970-01-01 14:15:12.793]

Tuple[TupleEntry: value=b, TupleEntry: value=0.3, TupleEntry: value=1970-01-01 14:15:12.793]

]

Processing State Changes

- **Generally you need to embed an engine to capture state changes.**
  - Hmmm, should we have a command line tool to query state?
- **Making use of state**
  - State can be periodically inspected (polled)
  - Callbacks can deliver state at end of a rule evaluation.
  - Rule actions can be defined using action tables.
- **A reader/writer lock is used over state.**
  - Engine acquires a write lock.
  - Reading waits for a read lock so that state is only read between evaluations.
- **An example**
  - Let's build a single engine application that receives messages and logs them into persistent state.
  - We'll use the ***tuple*** tool to send a message to the engine.
  - We'll use both a callback and direct state inspection.
  - See EngineTutorial.java in samples directory

First some rules

- **We'll define some rules in a Java String instead of reading them from a file.**

```
1) final static String rules =
2)    // This table is how the local runtime sends data into the engine to trigger evaluation
3)      "input inmsg(char[128] msg);\n"
4)    // This tables stores the messages we receive.
5)    + "persistent savedmsg(char[128] msg);\n"
6)    // This rule saves any messages we receive.  This is the state we'll capture.
7)    + "savedmsg(msg) if inmsg(msg);\n"
8)     // This rule saves messages we've already received.
9)     + "savedmsg(msg) if prev savedmsg(msg);\n"
10);
```

A Listener

- **IEvaluationListener is a simple interface with a single method to receive notification**
  - The state before and after rule evaluation is provided to the listener
- **A listener is defined to receive the end of evaluation events.**

```
1) public class EngineListener implements IEvaluationListener {
2)         // We'll use this flag to signal main() that evaluation has happened.
3)         public boolean evaluationComplete = false;

4)         public void endEvaluation(IRuleEngine e, TupleState beginState,
5)                                   TupleState endState, Object staticUserData) {

6)               // Get the messages we've received and stored in the savedmsg table.
7)               ITupleSet tset = endState.getTuples("savedmsg");
8)               System.out.println("\n\nEnd evaluation, saved messages...\n" + tset + "\n");

9)               // Flag that we've done an evaluation.
10)              evaluationComplete = true;
11)        }
12)  }
```

main()

```
1) // Create a descriptor for the name space used by the engine we're creating.
2) IApplicationDescriptor appDesc = new ApplicationDescriptor("EngineHelloWorld","tutorial");
3)
4) // Create the rule engine with the rules above.
5) DSMDefinition def = DSMParser.parse(new StringReader(rules));
6) DSMEngine dsmEngine = new DSMEngine(appDesc,  def);
7)
8) // Add a listener to get notified when a rule evaluation has completed.
9) EngineListener listener = new EngineListener();
10)dsmEngine.addListener(listener, null);
11)
12)// The engine must be started before anything can really be done with it.
13)dsmEngine.start();
14)
15)// Give the discovery system time to share its information with other engines.
16)System.out.print("Waiting for engine to register...");
17)Thread.sleep(5000);
18)System.out.println("...done waiting.");
```

main() (cont)

```
1) // Wait for the tuples from other engine or command line
2) // Listener will be called during this sleep(), if message is received.
3) System.out.print("Waiting for engine to receive tuples...");
4) while (!listener.evaluationComplete)
5)    Thread.yield();
6) System.out.println("...done waiting for tuples.\n");
7)
8) // Now we demonstrate sending input tuples from Java
9) System.out.println("Now adding tuples programmatically");
10)dsmEngine.addTuples("inmsg",new Tuple("World!"));
11)
12)// Print out all the tuples we received using direct state access.
13)ITupleSet savedMsgs = dsmEngine.getTuples("savedmsg");
14)for (ITuple t : savedMsgs) {
15)   String recvMsg = t.get(0).getValue();
16)   System.out.println("Received msg: " + recvMsg);
17)}
18)
19)// Shutdown the engine in an orderly manner.
20)dsmEngine.stop();
```

The *tuple* Command Line Tool

- **The tuple command line tool allows inserting and reading tuples to/from a running engine.**
  - Requires the name space and instance to identify the engine instance.
  - Inserts and targeted reads require name of the tuple/table
  - Inserts require the column values to set – only 1 row currently.
- **Usage**
  - tuple read <namespace> <instance-id> [<tablename>]
    - If the table name is not present, then all tables are read
  - tuple insert <namespace> <instance-id> <tablename> <column names and values>
    - <column names and values> is one of
      - A list of name/value pairs<column 0 name>=<value 0> <colu
      - The name of a file to read CSV values from. 1st line is a header specifying column names.
    - The column names must match those defined in the target table.
    - The types of values are inferred from the values (the first record in the case of CSV).mn 1 name>=<value 1> …
- **Examples**
  - tuple read router router1 path
    - reads the tuples in the 'path' table of instance 'router1' in the 'router' namespace
  - tuple insert router router1 path src=1.2.3.4 dest=5.6.7.8
    - - sends to application with id '1.2.3.4' in the 'router' namespace, a tuple insertion of tuple (1.2.3.4, 5,6.7.8) to the table named 'path'

Running EngineHelloWorld

- **Start main() (I do this from Eclipse)**
- **Run *tuple* to send tuples into the engine**
  - *tuple insert EngineHelloWorld tutorial inmsg msg=Hello*
- **main() produces the following:**

```
Waiting for engine to register.....done waiting.
Waiting for engine to receive tuples...

Listener called:
End evaluation, saved messages...
TupleSet:TupleSetDescriptor: appDesc=EngineHelloWorld/tutorial, name=savedmsg, [ColumnDescriptor: name=MSG, type=String, ColumnDescriptor: name=TOC__INTERNAL, type=Timestamp][
Tuple[TupleEntry: value=Hello, TupleEntry: value=1970-01-01 19:40:35.822]
]

...done waiting for tuples.  Now adding tuples programmatically

Listener called:
End evaluation, saved messages...
TupleSet:TupleSetDescriptor: appDesc=EngineTutorial/tutorial, name=savedmsg, [ColumnDescriptor: name=MSG, type=String, ColumnDescriptor: name=TOC__INTERNAL, type=Timestamp][
Tuple[TupleEntry: value=World!, TupleEntry: value=1970-01-01 03:26:29.768]
Tuple[TupleEntry: value=Hello, TupleEntry: value=1970-01-01 03:26:29.769]
]

State inspected directly:
Received msg: World!
Received msg: Hello
```

Action Tables

- **Action tables are used to define Java methods that are called at the end of a rule evaluation**
- **They have a minimum of two columns as follows:**
  - action my_actions(int unused, char[128] methodName);
    - The 1st column holds flag bits.
      - Bit 0: sets the actions to be call synchronously. Default is asynchronously.
    - The 2nd column defines the full name of the method, including the class name (e.g., com.ibm.watson.dsm.samples.RuleActionExample.action1).
    - The minimum signature of the action method is as follows:
      - public static void my_action(IRuleEngine r, TupleState begin, TupleState end);
- **Action tables may have additional columns. Extra column change the expected method signature thereby allowing these column values to be passed to the method. For example,**
  - action my_actions(int flags, char[128] methodName, **int x, char[128] y, double z**);
  - The system then expects the named method to have the following signature:
    - public static void action1(IRuleEngine r, TupleState begin, TupleState end, **Integer x, String y, Double z**);
    - Note that primitive types (int, double, etc) are mapped to the Java object types.

Action Tables Example

Taken from com.ibm.watson.dsm.samples.RuleActionExample.java in samples directory

- **First we define some rules that allow us to call a Java action method using 2 different patterns**
  - Pass in the method and parameters to call (see line 13 below).
  - Trigger a call to a specific method defined in the rules (see line 17 below)

```
1)  final static String rules =
2)     // An input table that gets copied into the action table
3)     // Note the x and y are column values that will be passed to the action method.
4)       "input to_call(int flags, char[128] method, char[128] str, int x, double y);\n"
5)
6)     // This input tuple, if seen, will trigger the setting of the action table with the action2 method.
7)     + "input trigger(int x);\n"
8)
9)     // The actions table defining  the method to call and its arguments.
10)    + "action actions(int unused, char[128] method, char[128] str, int x, double y);\n"
11)
12)    // Copy the values from to_call into the actions table, effectively calling what is passed with to_call
13)    + "actions(0,m,str,x,y) if to_call(s,m,str,x,y);\n"
14)
15)    // This sets values in the actions tables when the trigger tuple is seen.
16)    // Here we'll pass the trigger value and 2.0 to our action method action2.
17)    + "actions(0,\"com.ibm.watson.dsm.samples.RuleActionExample.action2\",\"Hello\",x,2.0) if trigger(x);\n";
```

Action Tables Example (cont)

- **Next we define our Java methods**
  - The method always receives IRuleEngine, and 2 TupleState objects.
  - Our rules define the action table with extra columns, so our method must expect those
  - The extra parameters ...`String str, Integer x, Double y)` match our extra columns.
    - They are required to match the extra columns defined in our action table that will call this method.
    - The values in the extra columns are converted to Java Objects before passing to the action method
      - int → Integer, char[] → String, double → Double.

```
public class RuleActionExample {
 …
 static void action1(IRuleEngine engine, TupleState begin, TupleState end,
                     String str, Integer x, Double y) {
     System.out.println("action1: str=" + str + ", x=" + x + ", y=" + y);
}

public static void action2(IRuleEngine engine, TupleState begin, TupleState end,
                     String str, Integer x, Double y) {
     System.out.println("action2: str=" + str + ", x=" + x + ", y=" + y);
}
 …
}
```

Action Tables Example (cont)

- **Next we write some Java code to load and drive the rules.**

```
1)   ...
2)   // Create the rule engine with the rules above.
3)   DSMDefinition def = DSMParser.parse(new StringReader(rules));
4)   DSMEngine dsmEngine = new DSMEngine(appDesc,  def);
5)
6)   // Start engine, otherwise nothing works.
7)   dsmEngine.start();

8)   // Load a tuple into to_call to set the action to call and the values to pass to it.
9)   String method = "com.ibm.watson.dsm.samples.RuleActionExample.action1";
10)  ITuple t1 = new Tuple(0, method, "string and values from tuple", 2, 3.0);
11)  // Send in the to_call tuple to the engine, which triggers an evaluation
12)  dsmEngine.addTuples("to_call", t1);

13)  // Now load a trigger tuple to set the action2 method to be called.
14)  t1 = new Tuple(10);
15)  // Send in the trigger tuple to the engine, again triggering an evaluation
16)  dsmEngine.addTuples("trigger", t1);
17)
18)  // Stop the engine cleanly.
19)  dsmEngine.stop();
```

Installation

- **Requirements**
  - 1.6 JVM
  - Windows or Linux
- **Installation**
  - Unzip release file (dsm-0.2-external.zip)
    - Windows: *mkdir c:\dev\dsm; cd c:\dev\dsm; unzip dsm-0.2-external.zip*
    - Linux: *mkdir ~/dsm; cd ~/dsm unzip dsm-0.2-external.zip*
  - Set DSM_HOME environment variable
    - Windows: *set DSM_HOME=c:\dev\dsm*
    - Linux: *export DSM_HOME=~/dsm*
  - Add DSM_HOME/bin to your execution path
    - Windows: *set PATH=c:\dev\dsm\bin;%PATH%*
    - Linux: *export PATH=~/dsm/bin:$PATH*
- **Configuration**
  - *dsm.properties* file provides some configuration of DSM platform and engine
  - Searched for in $DSM_PROPERTIES_FILE, current directory, $DSM_HOME/lib,
    - Executable prints out where it is being loaded from.
  - Most often used properties are those to configure logging, networking addresses and multicast/broadcast.

Thank You

IBM Research

Backup

T. J. Watson Research Center

Data Sharing Layer - Java Interface

- **Data Sharing**
  - *boolean shareData(ISharedData...data)*
    - Share a piece of data with associated platform meta data
    - All shared data has an associated name, like a key in a hash table, or table name in a DB.
  - *boolean unshareData(ISharedData...data)*
    - Remove a piece of data from the shared platform storage.
  - Platform storage implementations are interchangeable – persistent, relational, in-memory, etc.
- **Topology-based continuous queries**
  - *addRemoteDataListener(TopologyRelationship, IDataQuery, IDataAction)*
    - Listen for changes to shared data matching the query on the nodes with the given topological relationship and when such changes are detected, call the given action callback.
- **Topology-based continuous conditionals**
  - *addRemoteDataListener(TopologyRelationship, IDataCondition, IDataAction)*
    - Listen for changes to shared data referenced by the condition on the nodes with the given topological relationship and when such changes are detected and the given condition is met, call the given action callback.

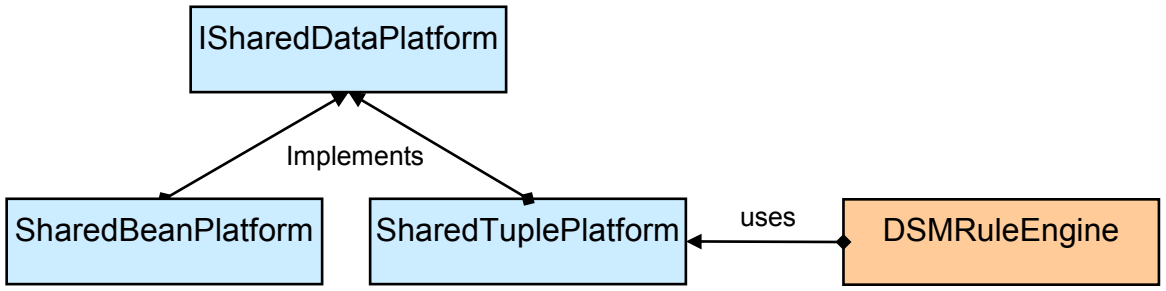
Data Sharing Layer - Java Interface (cont)

- **Topology-based synchronous queries**
  - *List<IQueryResult> queryRemoteData(TopologyRelationship, IDataQuery)*
    - Query remote shared data matching the query from the nodes with given topological relationship.

- **Topology-based synchronous conditionals**
  - *boolean isSatisfied(TopologyRelationship, IDataCondition)*
    - Apply the condition to the remote data referenced by the condition from the nodes with the given topological relationship and if it is satisfied, then return true.

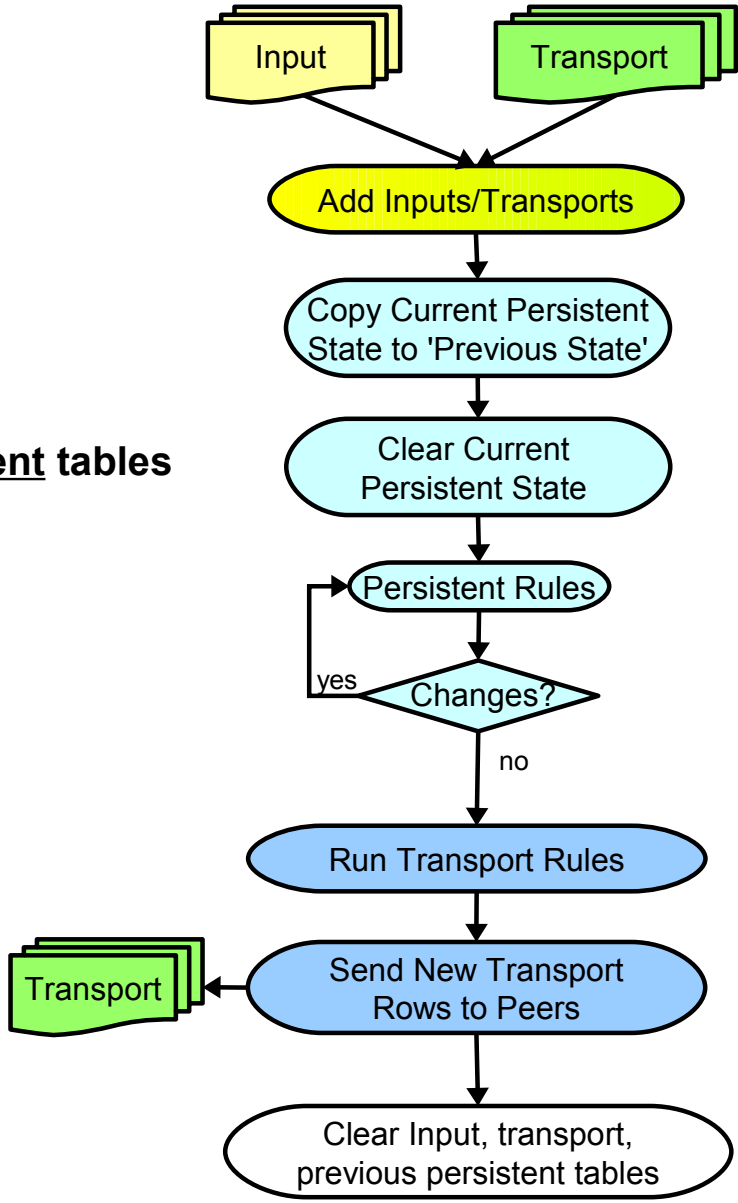
Data-specific Platform

- **Two implementations exist – others can easily be added as necessary**
  - SharedBeanPlatform – shares Serializable Java objects
  - SharedTuplePlatform – shares SQL table, row and column data
- **For each, there is a corresponding set of ISharedData-specific classes**
  - ISharedBean, IBeanAction, IBeanCondition, etc.
  - ITupleSet, ITupleAction, ITupleCondition, etc
- **DSMRuleEngine runs on the SharedTuplePlatform**



Rule Evaluation

1. **Append new input and/or transport rows**
2. **Copy current state to 'previous state'.**
3. **Clear persistent state**
4. **For each block of rules, until no more adds to persistent tables**
    1. Evaluate rules appending to persistent tables
5. **Evaluate rules on transport tables**
6. **Send new transport contents to destinations**
7. **Clear input, transport and previous persistent tables.**



An Application Node

- **Each node manages its own local data and controls what data are shared**
- **Shared data may be**
  - queried directly via remote peer applications
  - Continuously queried via remote peer subscriptions
  - modified in response to changes in local and remote data
- **Policy-enabled to control sharing/filtering of data**