WASP Manual

1 Introduction

WASP is a programming language for a subset of wireless sensor network applications that use stationary nodes to periodically sample data and transmit them to a base station. Data can be processed locally and aggregated at the network level.

When you start the test, three windows will be open for you on the desktop, as shown in Figure 1. Please keep them open during the test. If you close any of these windows by accident, please ask your instructor to reopen it. In Figure 1, the window at the top layer is the **WASP programming environment**, where your program is edited and run. The **Network** window shows program errors and sampled data for each node. The **Base station** window shows the data received at the base station. A network composed of four nodes is simulated in this test.

2 Concepts and Definitions

Here are some concepts and definitions we will use in this manual.

- **Identifiers** are composed of alphanumeric characters and underscores (_). The first character of an identifier must be a letter or an underscore. For example, *light_level* and *_sound_5* are legal identifiers, but *32light* is not. Identifiers are used for variable names. Variable names are case-sensitive. For example, *VAR* is a different variable from *Var*.

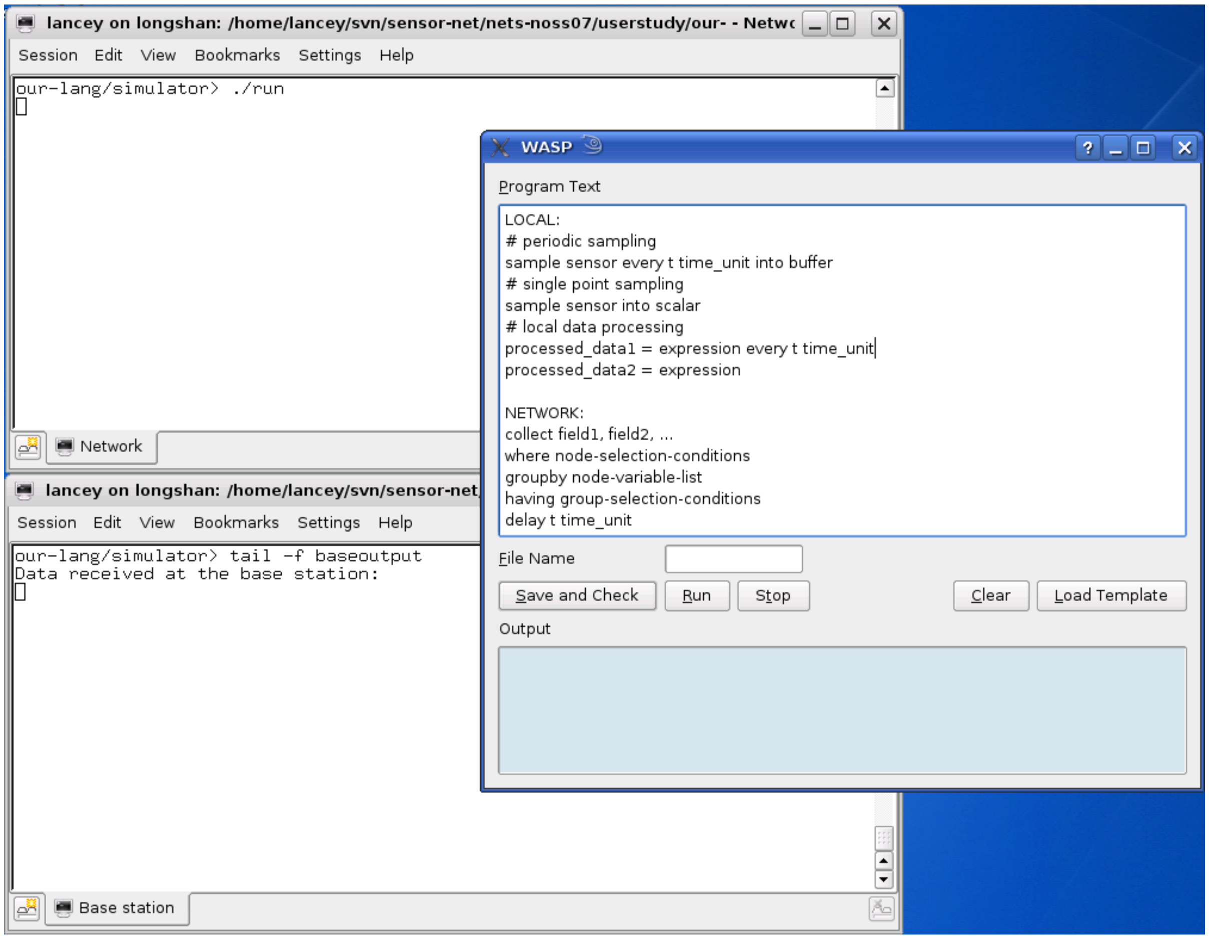
- **Numbers** are integers and decimals. For example, 32 and -5 are integers; -2.5 and 3.22E-2 are decimals.

- **Keywords** are certain words that cannot be used in programs to name variables. Keywords are case-insensitive: *sample* is the same as *SAMPLE* or *Sample*. WASP uses the following keywords:

  SAMPLE, COLLECT, EVERY, INTO, WHERE, GROUPBY, HAVING, DELAY,
  AND, OR, LOCAL, NETWORK, ms, sec, minute, hour, day

  Function names and sensor names are also keywords. They are listed in Table 1, Table 2, and Table 3.

- **Variables** are locations for storing data. Variable names must be identifiers. For example, $var =$ 40 defines a variable named *var* and sets its value to 40. Node identifier *nid* is a special variable with a value that is defined outside your program and stays unchanged. It is used to distinguish sensor nodes from each other. You can directly use it in your program. You are not allowed to assign values to *nid*.


lancey on longshan: /home/lancey/svn/sensor-net/nets-noss07/userstudy/our- - Netwo
Session Edit View Bookmarks Settings Help
Network
lancey on longshan: /home/lancey/svn/sensor-net/
Base station
WASP
Program Text
File Name
Save and Check
Run
Stop
Clear
Load Template
Output

```
our-lang/simulator> ./run
```

```
LOCAL:
# periodic sampling
sample sensor every t time_unit into buffer
# single point sampling
sample sensor into scalar
# local data processing
processed_data1 = expression every t time_unit
processed_data2 = expression

NETWORK:
collect field1, field2, ...
where node-selection-conditions
groupby node-variable-list
having group-selection-conditions
delay t time_unit
```

```
our-lang/simulator> tail -f baseoutput
Data received at the base station:
```

Figure 1: WASP Programming and simulation environment.

- **Arithmetic expressions** are composed of variable names, binary operators, and parenthesis. A binary operator can be +, -, *, or /. An operand can be an identifier or a number. For example, $(mysound * 2 + 100)/10$ is an arithmetic expression.
- **A comparison operation** compares the values of two variables or compares the value of a variable with a number. It takes the form: *identifier operator identifier* or *identifier operator number*. The operator can be $>$ (greater than), $<$ (less than), $==$ (equal), $>=$ (greater than or equal), $<=$ (less than or equal), or $<>$ (not equal). The result of an comparison operation can either be true or false. For example, $mysound > 10$ is true if mysound equals 20.
- **Functions** take a fixed number (zero or more) of parameters and return values. You can use a function in the following way: function_name(argument_list). The argument list is a sequence of arguments separated by ",". An argument can be an identifier or a number.
- **Comments** are used to insert notes in one's program. They do not affect functionality. In WASP, "#" indicates the start of a comment, which extends to the end of a line.

Sensor Name	light	temperature	humidity	pressure	sound

Table 1: Sensors.

Function name	Arguments	Returned result
MAX_TIME	*buffer[m:n]*	the maximum value in *buffer[m:n]*
MIN_TIME	*buffer[m:n]*	the minimum value in *buffer[m:n]*
AVG_TIME	*buffer[m:n]*	average of the values in *buffer[m:n]*
SUM_TIME	*buffer[m:n]*	sum of the values in *buffer[m:n]*

Table 2: Node-level Functions.

3 WASP Programming Language Construct

A program is composed of two segments. The node-local code segment is initiated with the keyword "local:". It describes node local behaviors, such as sampling and local data processing. The network-level code segment is initiated with the keyword "network:". It describes how data are aggregated at the network level and gathered at the base station. A template for WASP programs is given below. Upper-case words are commands; you don't need to capitalize them when you write your program. Lower-case words are descriptions of parameters at the corresponding location; you need to replace them with the variables, functions, and expressions you want to use. Detailed descriptions are in Section 3.1 and Section 3.2.

```
LOCAL:
SAMPLE sensor EVERY t time_unit INTO buffer
SAMPLE sensor INTO scalar
processed_data1 = function(arg1, arg2, ...) EVERY t time_unit
processed_data2 = function(arg1, arg2, ...)
processed_data3 = arithmetic_expression EVERY t time_unit
processed_data4 = arithmetic_expression

NETWORK:
COLLECT field1, filed2, ...
WHERE  node-selection-conditions
GROUPBY node-variable-list
HAVING group-selection-conditions
DELAY t time_unit

COLLECT field3, field4, ...
WHERE  node-selection-conditions
GROUPBY node-variable-list
HAVING group-selection-conditions
DELAY t time_unit
```

Function name	Arguments	Returned result
MAX	*attribute*	the maximum value of *attribute* in each group
MIN	*attribute*	the minimum value of *attribute* in each group
AVG	*attribute*	the average of *attribute* in each group
SUM	*attribute*	the sum of *attribute* in each group

Table 3: Network-level Functions.

3.1 Node-Local Code

The node-local code segment contains two types of functionalities.

- **Data sampling**: Specify the type of and frequency of sensor data sampling.

  Syntax:

  SAMPLE sensor EVERY t time_unit INTO buffer

  SAMPLE sensor INTO scalar

  Sensor represents what type of sensor data to be sampled. It is selected from a given list as shown in Table 1. *t* is the time interval between two samples; it must be a positive integer. *time_unit* is a time unit. It can be *ms, sec, minute, hour,* or *day.* The variable *buffer* is where sampled data are stored. You can view it as a infinite array that stores the sampled data in a time series. Buffer[0] represents the most recent sample. Buffer[n] represents the data sensed n samples ago. Buffer[m:n] indicates the data gathered since n samples ago until m samples ago, which contains $n - m + 1$ data elements. For example, buffer[0:9] returns the most recent 10 samples. As shown in the second syntax, if you don't specify the sampling period, it will be a single point sampling, i.e., sample only once.

- **Data processing**: Specify how the raw sensed data are processed to generate other data. This may be used for data interpretation, unit conversion, local event detection, etc.

  Syntax:

  new_data = arithmetic_expression

  new_data = arithmetic_expression EVERY t time_unit

  new_data = function(argument1, argument2, ...)

  new_data = function(argument1, argument2, ...) EVERY t time_unit

  Parameter *t* specifies how frequently the value of *new_data* will be recomputed. *Function* is selected from a library of built-in functions listed in Table 2. These functions are aggregation functions used in node-level code. They aggregate data across time on each single node instead of aggregating data across nodes.

3.2 Network-Level Code

The network-level code segment lets programmers view the entire sensor network as a whole and use network-level operations to extract the data they want based on local data. Local data must be variables defined in local level code segment. A list of supported operations follows.

- **COLLECT field1, field2...**: Collect data or aggregated data named field1, field2... from the sensor network at the base station. A field can either be a node-level variable that are defined in the node-level code or aggregated data across nodes in the network. Aggregation operations apply to a set of nodes that are grouped together via the GROUPBY operation. Network-level aggregation functions are shown in Table 3. These functions aggregate data across different nodes. They are used in the network-level code segment. Only those nodes that are selected with WHERE and HAVING commands are involved in the operation.
- **WHERE node-selection-conditions**: Select nodes that satisfy the specified conditions. The conditions may contain multiple comparison operations, which are connected with **AND** or **OR**. E.g., WHERE nodeid > 10 AND light < 50. If every node will be selected, you can eliminate this statement.
- **GROUPBY node-variable-list**: Group nodes so that nodes in each group have the same values of all the variables listed. If you do not need aggregation, you can eliminate this statement.
- **HAVING group-selection-conditions**: Select groups that satisfy the specified conditions. E.g., HAVING AVG(temperature) > 100 selects groups with average temperatures higher than 100. Again, multiple comparison operations can be combined with **AND** or **OR**. If you do not group nodes or you want to collect data from all groups, you can eliminate this statement.
- **DELAY t time_unit**: Specify how fast should data be gathered. Parameter *t* means the maximum elapsed time since the data is generated until data arrive at the base station. If you just require the data to be eventually gathered with arbitrary delay, eliminate this statement.

The network-level code segment can contain multiple COLLECT commands. Each COLLECT needs to be followed by WHERE, GROUPBY, HAVING, and DELAY commands in order, yet any of them can be optional depending on the application. A COLLECT command will be executed with the shortest period of the collected data. If you sample different types of data with different periods, you may want to have one collect command for each of them in order to collect the data at a frequency consistent with its sampling frequency.

4 An Example Using WASP

In this section, we will show an example of how to write a sensor network program in WASP.

Assume we want to deploy sensor nodes around a redwood tree to study the microclimate of the redwood tree. The nodes are able to sense temperature and barometric pressure. The height level of a node can be computed from the pressure by *pressure/100 + 2*. Sensor nodes are static, so we only need to compute nodes' height levels once. Write a program to sample the average light at each height level and transfer this value to the base station. Light will be sampled every 1 second on each node. Each node first averages its own samples within 4 seconds, then the averaged data across nodes are averaged within height levels.

```
local:
  SAMPLE light EVERY 1 sec INTO lightbuf
  SAMPLE pressure INTO mypressure
  height = mypressure / 100 + 2
  my_avg_light = AVG_TIME(lightbuf[0:3]) EVERY 4 sec
```

```
network:
  COLLECT height, AVG(my_avg_light)
  GROUPBY height
```

5 WASP Programming Environment

The programming environment, as shown in Figure 1, provides tools to check grammar errors in your program and simulate its functionality. The steps to compromise your code, check it, and simulate it follow.

1. Edit your program in the **Program Text** field. You can modify the template, or delete the template by clicking the **Clear** button and edit your program from the scratch. The programming template can be loaded to the **Program Text** field by clicking on the **Load Template** button. Note that doing this will overwrite the code in the **Program Text** field.

2. Enter the name for your program in the **File Name** box, and click the **Save and Check** button to save your program into a file and check for grammar errors. If there is no grammar error, you should see the following line printed in the **Network** window:

   Program check done. No syntax error found.

   If there are syntax errors, the error message will be printed in the **Network** window.

3. Click the **Run** button to execute your program in a simulated sensor network. You can check the results in the **Network** window and the **Base station** window. If you want to stop the simulation, click the **Stop** button.

Figure 2 shows the simulation output of the example application in Section 4 in the **Network** window and the **Base station** window. The **Network** window shows the sensor readings for each node at each time stamp. Data values in the "[ ]" are listed in the order of the node id. For example,

Time 0 s, sample light : [618, 814, 284, 1009]

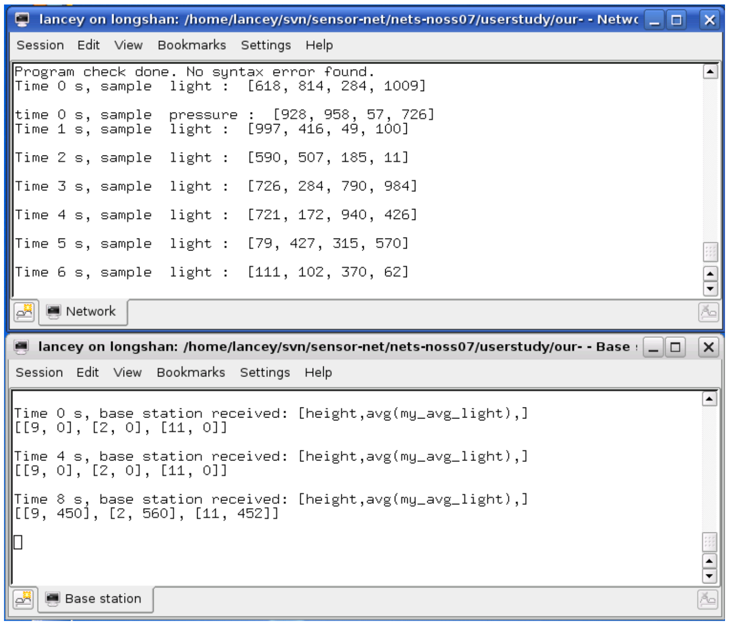
indicates that at time 0, node 0 samples light level 618, node 1 samples light level 814...

The **Base station** window shows the data received at the base station. The data values in the inner square brackets follows the format in the preceding line. For example,

Time 8 s, base station received: [height,avg(my_avg_light),]

[[9, 450], [2, 560], [11, 452]]

indicates that at time 8 s, the base station received 3 groups of data. The first group has height value 9 and average light value 450, the second group has height value 2 and average light value 560, ...



lancey on longshan: /home/lancey/svn/sensor-net/nets-noss07/userstudy/our- - Netwc

```
Program check done. No syntax error found.
Time 0 s, sample  light :  [618, 814, 284, 1009]

time 0 s, sample  pressure :  [928, 958, 57, 726]
Time 1 s, sample  light :  [997, 416, 49, 100]

Time 2 s, sample  light :  [590, 507, 185, 11]

Time 3 s, sample  light :  [726, 284, 790, 984]

Time 4 s, sample  light :  [721, 172, 940, 426]

Time 5 s, sample  light :  [79, 427, 315, 570]

Time 6 s, sample  light :  [111, 102, 370, 62]
```

lancey on longshan: /home/lancey/svn/sensor-net/nets-noss07/userstudy/our- - Base :

```
Time 0 s, base station received: [height,avg(my_avg_light),]
[[9, 0], [2, 0], [11, 0]]

Time 4 s, base station received: [height,avg(my_avg_light),]
[[9, 0], [2, 0], [11, 0]]

Time 8 s, base station received: [height,avg(my_avg_light),]
[[9, 450], [2, 560], [11, 452]]
```

Figure 2: Simulation output of example.