Agent-Based Workbench for On-the-Fly Sensor-Data Analysis [*]

Munehiro Fukuda

Computing & Software Systems
University of Washington, Bothell,
18115 NE Campus Way, Bothell, WA 98011
Email: mfukuda@u.washington.edu

Abstract

The emergent popularity of wireless sensor networks and cloud-computing services brings us new opportunities of integrating them for performing on-the-fly sensor-data analysis with as many computing resources as needed. Yet, users are required to integrate sensor networks and cloud services manually, and even to redesign their data-analyzing software for its parallelization. To alleviate these problems, we are developing an agent-based workbench that consists of three independent but interlinked software tools: (1) the AgentTeamwork-Lite job coordinator, (2) the MASS middleware library, and (3) the Connector elastic data channel. AgentTeamwork-Lite has each computing node exchange resource information with neighbors, and deploys a job with a mobile agent that searches for idle nodes. MASS allows programmers to view their applications as a series of interaction among autonomic computation entities that are dynamically mapped to a cluster of multi-core machines. Connector separates descriptions of network address, communication protocol, and data-sampling conditions from applications. It automatically streams and hands off sensor data to migrating jobs. This paper presents an overview of our workbench and its applicability to orchard business.

1 Introduction

The emergent popularity of wireless sensor networks and cloud services have brought us new opportunities of on-the-fly sensor-data analysis in the cloud. For instance, consider orchard business. Frost protection is a big concern among tree-fruit growers. Temperature sensor networks are vital to observe every overnight transition of orchard air temperature, which then can provide growers with accurate information to protect their crops. Temperature prediction can be practicalized using the following computation scenario: estimating the temperature of sensor-uncovered areas with temperature interpolation [4]; predicting the overnight temperature of each area with artificial neural network [8] or empirically-formulated polynomials [9]; and storing sensor data in databases for future use in prediction. However, the problem is that accurate and on-the-fly prediction requires substantial computing resources.

Cloud computing could facilitate the above computation scenario. For instance, Amazon offers various cloud services: the EC2 cloud compute service to run the prediction programs, OpenMP as well as Open MPI to parallelize the programs for on-the-fly computation, and the S3 storage service to record sensor data. However, users will quickly encounter the following problems: where and how frequently their temperature prediction program should run from an economical viewpoint; how sensor data can be automatically forwarded to their program; and how the program can be parallelized to run at real-time speed. In reality, few users have such advanced skills that they can integrate their sensor networks and cloud services into a temperature prediction system.

Our main goal is to alleviate these hurdles and glue individual cloud services together with an agent-based workbench that consists of three independent but interlinked software tools: (1) the AgentTeamwork-Lite job coordinator, (2) the MASS middleware library, and (3) the Connector elastic data channel. AgentTeamwork-Lite has each computing node exchange resource information with neighbors repeatedly to construct a global computing resource potential field, and deploys a job with mobile agent that rolls down a steep slope of the potential field to launch a job at the best node. MASS facilitates entity-based programming in that an appli-

[*]This research is being conducted with partial support from UW Provost International Grants, Faculty-led Program

cation is viewed as a series of interaction among autonomic computation entities with independent code. These entities are dynamically mapped to threads running on a cluster of multi-core machines, and their communication is automatically translated into inter-process/thread communication. Connector separates descriptions of network address, communication protocol, and data-sampling conditions from applications. It automatically streams and hands off sensor data to migrating jobs. This paper presents an overview of our workbench and its applicability to orchard business.

2 Issues

A typical platform for realizing on-the-fly sensor-data analysis includes: (1) inputs from wireless sensor networks, sensor Web portals, and GIS websites; (2) computing facilities such as a commercial compute cloud or a business/academic partner's cluster; and (3) outputs presented onto a user-local graphics system and/or saved into local/remote file servers. On top of this platform, users would have to build their own system for sensor-data analysis, by choosing grid and cloud services that provide: (1) *job coordination* to deploy a job to appropriate computing nodes, (2) *job parallelization* to run a job in parallel for real-time analysis, and (3) *data delivery* to provide a job with sensor data in a speedy and fault-tolerant manner.

Job coordination: OpenPBS and Condor [2] are job deployment (and migration) tools, widely available for clusters and compute clouds such as Amazon EC2. While they are optimized to schedule whole computing nodes, it requires users to identify remote computing resources *a priori*. In particular for a parallel job execution, users may still need to specify resource requirements in configuration files. However, unsophisticated users need autonomic job deployment and migration where they are not required to identify remote computing resources or specify resource requirements.

Job parallelization: Sensor-data analysis is in most cases computation intensive, and coded in a model-based simulation [3] that uses formula-based, spatial, and/or multi-agent models. For real-time analysis, it should be parallelized with MPI, multi-threading, or parallel simulators. However, most model designers will experience a steep learning curve of the parallel-computing concepts.

Data delivery: High-speed and fault-tolerant data deliveries are facilitated with grid-computing technologies such as GridFTP [6]. Furthermore, the publisher/subscriber model has been proposed for sensor-cloud research [7] to inform data-analyzing programs of changes in sensor data. However, unless sensor data gets prepared as local files, applications need to be hard-coded with URLs, (i.e., underlying communication protocols and IP addresses) to retrieve sensor data. Moreover, a sensor-side publisher must detect changes in sensor data, maintain a table of subscribers, and hand off data to subscribers that may be nomadic over the Internet.

In summary, users still need to perform time consuming and knowledge intensive manual operations to orchestrate these services. This is our motivation to develop an agent-based workbench for on-the-fly sensor-data analysis.

3 System Design

This section explains our Java-based specification and design of the software components: AgentTeamwork-Lite, MASS, and Connector.

3.1 AgentTeamwork-Lite Job Coordinator

AgentTeamwork-Lite use the following two autonomic-computing approaches: (1) *bottom-up self-organizing resource management:* each computing node exchanges its resource information with the neighboring nodes periodically to maintain a partial view of a global computing-resource potential field that is virtually and dynamically built over all the nodes; and (2) *top-down self-adapting job execution:* a user job is submitted with a mobile agent that spawns child agents rolling down on a steep slope of the computing-resource potential field where the nearest hole corresponds to the best computing resources physically nearby the user. As illustrated in Figure 1, AgentTeamwork-Lite consists of six execution layers. Layers 1 - 3 are involved in bottom-up self-organizing resource management, whereas layers 4 - 6 are in charge of top-down self-adapting job execution.

1. **Layer 1: UDP-broadcast space:** The lowest layer is a TCP-link-assisted inter-segment UDP-broadcast space. Since UDP broadcast is limited to within a single segment, additional administrative support, such as IGMP, is necessary to allow broadcasting across multiple segments. Our design facilitates application-level inter-segment UDP broadcast by establishing a secured TCP link between representative nodes of each segment and allowing relaying of intra-segment UDP-broadcast messages among the segments.
2. **Layer 2: UWAgents:** The second layer is the UWAgents mobile-agent execution platform [5]. A

separate daemon process runs at each node exchanging agents with other nodes and running their code.

3. **Layer 3: Computing-resource potential field:** The third layer consists of Potential-Field Agents (PFAgents). Launched at each node, a PFAgent periodically measures the latest performance of its local computing resources including CPU power, memory space, disk size, network bandwidth, and their current availability. All of this information is recorded in each PFAgent's internal resource table and is broadcast in a UDP message within the local network segment and relayed to remote segments. Each PFAgent uses this information to guide a user process to the best nodes.
4. **Layer 4: Commander and sentinel agents:** The fourth layer is the commander and sentinel agents. The commander agent is launched with a user program, arguments, and resource requirement. It spawns the sentinel agent that repeats querying the local PFAgent for a site to run its job and actually migrating to that site. Once the agent realizes that the current and its neighboring nodes satisfy the user request, it generates an application-dependent configuration file (such as an MPI host file or a MASS configuration file) and starts a user job. During the job execution, the sentinel agent is responsible for periodically contacting the local PFAgent for resource-monitoring purposes and migrating the job to a less-loaded site if necessary.
5. **Layer 5: Middleware Libraries:** The fifth layer corresponds to middleware libraries such as MPI, OpenMP, and MASS for running a user application in parallel. As far as the sentinel agent supports their configuration file (e.g., an MPI host file), there is no need to change the libraries.
6. **Layer 6: Applications:** The sixth layer is where sensor-data analyzing programs run.

The key is interaction between a sentinel and a PFAgent. Each PFAgent i periodically receives resource information with its N neighbors and computes its cpu_rank_i, $mem\_free\_rank_i$, $mem\_load\_rank_i$, and $net\_bandwidth\_rank_i$ as its occupancy in the entire N summation of each performance measure. As shown in Formula 1, the comprehensive $rank_i$ is calculated as a sum of each measure weighted with its predefined priority. To move a job from the *current* to a *next* computing node, a sentinel agent uses Formula 2 where *hop_overhead* needs to be empirically obtained and *job_size* may be fixed to some jobs such as temperature analysis.

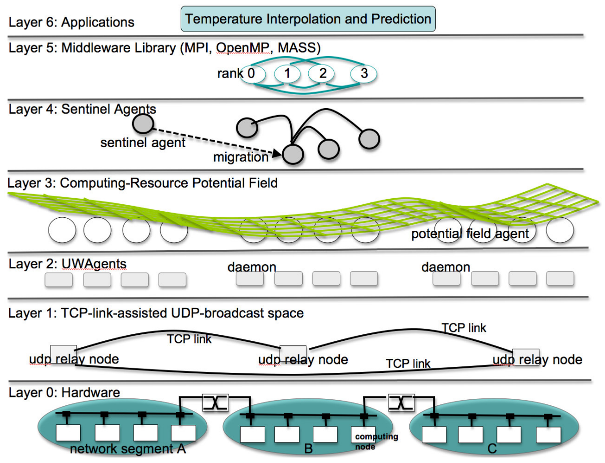


Figure 1. System layers

$$\begin{aligned} rank_i &= cpu\_rank_i \times w_1 + mem\_free\_rank_i \times w_2 \\ &+ mem\_load\_rank_i \times w_3 \\ &+ net\_bandwidth\_rank_i \times w_4 \qquad (1) \\ job\_size &\geq hop\_overhead + job\_size \times \frac{rank_{current}}{rank_{next}} \qquad (2) \end{aligned}$$

3.2 MASS Library for Multi-Agent Spatial Simulation

The MASS library (layer 5 in Figure 1) facilitates entity-based programming by allowing an application to be viewed as repetitive interaction among computation entities, each describing independent behavior in its own code, eliminating the need in for-loops to scan all entities. As a result, the library enables the following two features: (1) *no awareness of processes, threads, and their communication:* the library takes care of entity-to-process/thread mapping and thus automatically transforms inter-entity communication into underlying inter-process/thread communication; and (2) *dynamic load balancing:* since there is no more need to index entities in for-loops, entities are freely distributed and reallocated to any computing nodes and CPU cores.

3.2.1 Execution model

Places and *Agents* are keys to the MASS library. *Places* is a multi-dimensional array of elements that are dynamically allocated over a cluster of multi-core computing nodes. Each element is called a *place*, is pointed to by a set of network-independent array indices, and is capable of exchanging information with any other

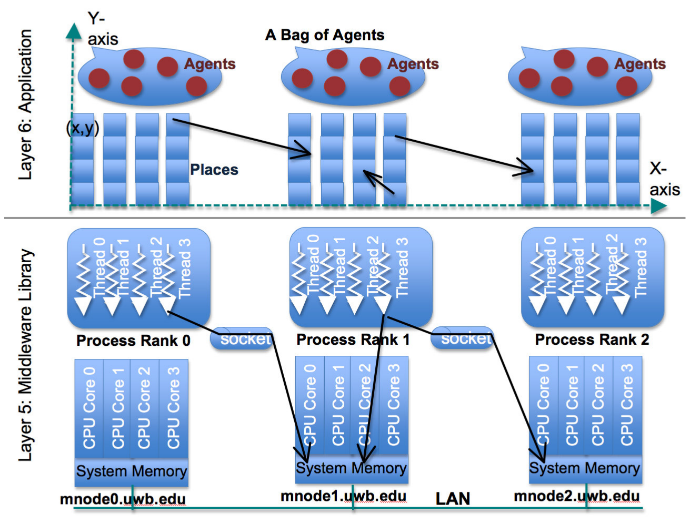


Figure 2. Parallel execution with MASS library

*place*s. *Agents* are a set of execution instances that can reside on a *place*, migrate to any other *place*s with array indices (thus as duplicating themselves), and interact with other *agent*s and *place*s.

As shown in Figure 2, parallelization with the MASS library uses a set of multi-threaded communicating processes that are forked over a cluster and are connected to each other through ssh-tunneled TCP links. The library spawns the same number of threads as that of CPU cores per node. Those threads take charge of method call and information exchange among *place*s and *agent*s in parallel. *Place*s are mapped to threads, whereas *agent*s are mapped to processes. Unless a programmer indicates his/her *place*-allocation algorithm, the MASS library partitions *place*s into smaller stripes in vertical, each of which is statically allocated to and executed by a different thread (static scheduling). Contrary to *place*s, *agent*s are allocated to a different process, based on their proximity to the *place*s that this process maintains, and are dynamically executed by multiple threads belonging to the process (dynamic scheduling).

3.2.2 Programming Style

A user designs a behavior of a *place* and an *agent* by extending the *Place* and *Agent* base classes respectively. S/he can populate them through the *Places* and *Agents* classes. Actual computation is performed between *MASS.init( )* and *MASS.finish( )*, using the following methods.

Places Class

- ***public Places(int handle, [String primitive,] className, Object argument, int size)*** instantiates a distributed array with *size* from *className* or a *primitive* data type as passing an *argument* to the constructor.
- ***public Object[] callAll(String functionName, Object[] arguments)*** calls the method specified with *functionName* of all array elements in parallel as passing *arguments[i]* to element[i], and receives a return value from it into *Object[i]*.
- ***public Object[] callSome(String functionName, Object[] argument, int... index)*** calls a given method of one ore more selected array elements in parallel. If *index[i]* is a non-negative number, it indexes a particular element, a row, or a column. If *index[i]* is a negative number, say $-x$, it indexes every x element.
- ***public void exchangeAll(int handle, String functionName, Vector<int[]> destinations)*** calls from each of all elements to a given method of all destination elements, each indexed with a different *Vector* element. Each vector element, say *destination[]* is an array of integers where *destination[i]* includes a relative index (or a distance) on the coordinate i from the current caller to the callee element. The caller passes its *outMessage[]* data member to the callee as a set of arguments, and receives return values in its *inMessage[]*.
- ***public void exchangeSome(int handle, String functionName, Vector<int[]> destinations, int... index)*** calls from each of the elements indexed with *index[]*. The rest of the specification is the same as *exchangeAll( )*.

Agents Class

- ***public Agents(int handle, String className, Object argument, Places places, int population)*** instantiates a set of agents from *className*, passes the *argument* to their constructor, associates them with a given *Places* matrix, and distributes them over these places, based on the *map( )* method that is defined within *Agent*.
- ***public void manageAll( )*** updates each agent's status, based on its latest calls of *migrate( )*, *spawn( )*, *kill( )*, *sleep( )*, *wakeup( )*, and *wakeupAll( )*. These methods are defined in the *Agent* class and invoked from other functions through *callAll( )* and *exchangeAll( )*.

Figure 3 shows a simple temperature prediction program that creates a virtual space of a meshed orchard (line 8); sets communication destinations (lines 11-25);

and performs a cyclic analysis (line 18-20). In each simulation cycle, each orchard cell exchanges local temperature with its neighbors (line 19) and updates its own temperature (line 20).

```
1   import MASS.*;
2   import java.util.Vector;
3
4   public class TemperatureAnalysis {
5     public static void main(String[] args) {
6       int size = 100, maxTime = 1000;
7       MASS.init(args);
8       Places ochard=new Places(1, "Orchard", null, size, size);
9
10      // define the four neighbors of each cell
11      Vector<int[]> neighbors = new Vector<int[]>( );
12      int[] north = {0, -1}; neighbors.add(north);
13      int[] east  = {1,  0}; neighbors.add(east );
14      int[] south = {0,  1}; neighbors.add(south);
15      int[] west  = {-1, 0}; neighbors.add(west );
16
17      now go into a cyclic simulation
18      for (int time = 0; time < maxTime; time++) {
19        streets.exchangeAll(1,Ochard.exchange,neighbors);
20        streets.callAll(Orchard.update);
21      }
22      MASS.finish( );
23  } }
```

Figure 3. User code using MASS library

3.3 Connector: Data Channels Redirectable to Remote Sensors, Files, and X servers

As shown in Figure 4, Connector makes it possible for a nomadic user program to exchange data with remote sensors, file servers, and a user console as if it operated onto local files, the standard input/output, and the local X server. It consists of three components: (1) *Connector-API:* a program-side I/O and graphics package, (2) *Connector-GUI:* a user-side GUI, and (3) *Sensor Server:* a sensor-side sensor publisher.

Connector-API allows a remote user program to behave as various protocol clients (including FTP, HTTP, and X windows) to access remote data through the major Java classes such as FileInputStream, FileOutputStream, Frame, and Graphics. File-to-URL mapping is no longer hard-coded in a user program but is rather specified in a file map, using a quadratic notation of: $(name, URL[, interval, extract])$, where each parameter represents (1) a file *name* used in a user program, (2) the corresponding *URL* including a user account and password, (3) a repetitive access to a given Web site every *interval* seconds, and (4) only text data *extract*ed from the Web. Figure 5 shows an example of file-to-URL mapping. The three files: sensor1, file2.txt, and file3.txt, all named in a user program, are respectively retrieved from the corresponding sensor, FTP, and HTTP servers. In particular, file3.txt provides the

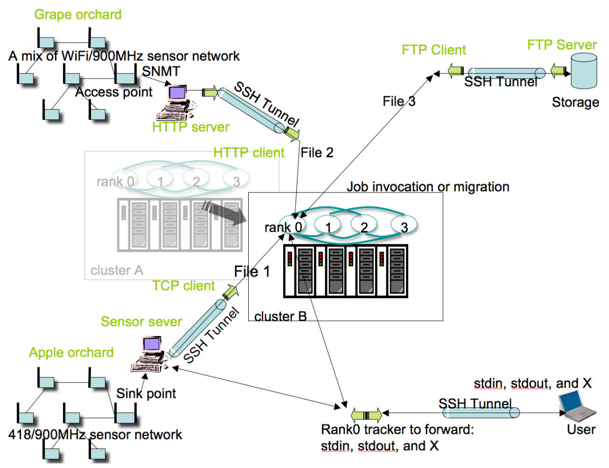


Figure 4. Elastic stream-oriented channels

user program with data items that are repeatedly read from the same website every 5 seconds.

sensor1	sftp://account:password@hercules.uwb.edu/sensor1		
file2.txt	ftp://account:password@ftp.tripod.com/temperature.txt		
file3.txt	http://www.weather.com/today/Bohell+WA+98011	5	extract

Figure 5. File-to-URL mapping in Connector

Figure 6 is an example user program that takes the file map shown in Figure 5. It first starts a Connector daemon thread (line 6). While the user program writes to the standard output, reads *sensor1*, and writes *file2* locally, in background the daemon forwards "Recording..." to a remote user's console (line 7), connects to *hercules.uwb.edu/sensor1* (line 8), contacts with *ftp.tripod.com* as an FTP client (line 10), and transfers data from the sensor to the ftp server (lines 12-13). The Connector daemon is capable of behaving as an FTP, HTTP, and X client.

```
1   import Connector.*;
2   import java.util.Scanner;
3
4   public class TemperatureRecording {
5     public static void main( String[] args ) {
6       Connector System = new Connector("file.map");
7       System.out("Recording...");
8       FileInputStream in=new FileInputStream("sensor1");
9       Scanner input=new Scanner(in);
10      FileOutputStream out=new FileOutputStream("file2");
11      DataOutputStream output=new DataOutputStream(out);
12      while(input.hasNextLine( ))
13        output.writeUTF(input.nextLine( ));
14      System.close( );
15  } }
```

Figure 6. Code using Connector-API

Connector GUI runs on a user-local machine to facilitate GUI by forwarding keyboard/mouse inputs to and by receiving monitor outputs from a remote user program. It is also capable of transferring files between user-local disks and each user program. To provide these features, the GUI runs as a TCP server to keep track of a nomadic user program by accepting its connection request upon a migration; scrutinizes each message for its data delivery; and works as an X proxy client.

Sensor server runs on a sensor-network sink node in charge of (1) managing all its sensor devices, (2) behaving as an FTP server to make all sensor devices accessible as files from applications, (3) detecting changes in sampled data, and (4) handing off active connections to nomadic jobs. For these purposes, as shown in Figure 7, it receives commands to add, delete, change, and detect a sensor's IP address, file name, and data-sampling conditions, all from the Connector GUI or its configuration file

add	192.168.15.21	sensor1
add	192.168.15.22	sensor2
detect	sensor2 <= 33.6	absolute

Figure 7. Sensor-to-File mapping in Sensor Server

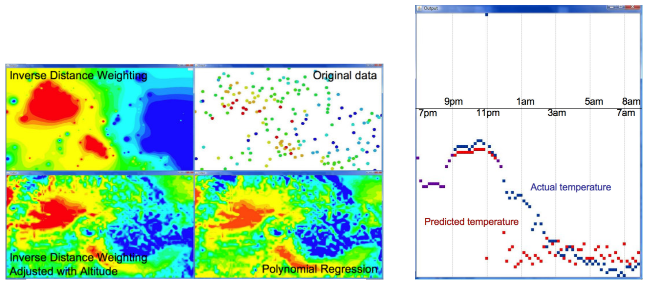
4 Application to Orchard Business

Temperature prediction is the key to frost protection. It takes two steps: (1) temperature interpolation and (2) hourly-based prediction.

Temperature interpolation is a software technique to estimate temperature on each divided cell of a given land with N sampled temperature data. This is important, because the scale of a wireless sensor network installed on an orchard is generally too limited to perfectly measure the air temperature surrounding all the crops. We use inverse distance weighting [4] that computes each cell's air temperature T, based on Formula 3 where d_i is a distance from sensor i to this cell; t_i is air temperature sampled by sensor i; and p is normally 2. The altitude should be considered for more accuracy, in which case we estimate each cell's altitude using the same algorithm, obtain the actual altitude, and adjust T with the difference between estimated and actual altitude data.

$$T = \sum_{i=1}^{N} \left(\frac{d_i^{-p}}{\sum_{j=1}^{N} d_j^{-p}}\right) t_i \qquad (3)$$

Two air-temperature prediction algorithms are available to use from University of California, Davis



(a) Interpolation (b) Prediction

Figure 8. Temperature analysis

and University of Georgia respectively [9, 8]. The former uses temperature data T_i observed at two hours after sunset, predicts the minimum temperature T_p by applying the UC Davis-developed polynomial to the observed temperature, and plots hourly predicted temperature from T_i down to T_p using another polynomial [9]. The latter uses an artificial neural network (ANN) model and predicts temperature from one to 12 hours ahead by training ANN with seasonal parameters and weather forecast information [8].

This series of temperature interpolation and prediction can be parallelized as a MASS application that instantiates a Place object to model an orchard and to maintain temperature for each cell as shown in Figure 3. Connector's sensor server runs on the sink node of a wireless sensor network to filter and to send sensor data to the MASS program. The Connector daemon thread receives and pumps temperature data to the MASS program as well as accumulates sampled/predicted temperature into a file server. The Connector GUI presents to the corresponding grower's notebook a temperature map and a prediction graph as shown in Figures 8-(a) and 8-(b).

5 Conclusions

This paper focused on integrating wireless sensor networks into grid and cloud computing systems, and presented our agent-based workbench running on top of these computing systems for on-the-fly sensor-data analysis, particularly orchard temperature prediction. Our next step is to complete our development and to deploy the system to our business partners [1, 10].

References

[1] AgComm. http://agcomm.net/.
[2] Condor Project. http://www.cs.wisc.edu/condor/.

[3] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *The VLDB Journal*, Vol.14(No.4):417–443, November 2005.

[4] Fred C. Collins Jr. A Comparison of Spatial Interpolation Techniques in Temperature Estimation. In *Third International Conference/Workshop on Integrating GIS and Environmental Modeling*, pages in CD–ROM, Santa Fe, NM, January 1996. NCGIA.

[5] M. Fukuda and D. Smith. UWAgents: A mobile agent system optimized for grid computing. In *Proc. of the 2006 International Conference on Grid Computing and Applicaitons – CGA'06*, pages 107–113, Las Vegas, NV, June 2006. CSREA.

[6] Globus Allianace. http://www.globus.org.

[7] M. M. Hassan, B. Song, and E.-N. Huh. A framework of sensor-cloud integration opportunities and challenges. In *Proc. of the 3rd International Conference on Ubiquitous Information Management and Communication*, pages 618–626, Suwon, Korea, January 2009. ACM.

[8] B. A. Smith, G. Hoogenboom, and R. W. McClendon. Artificial neural networks for automated year-round temperature prediction. *Computers and Electronics in Agriculture*, Vol.68(Issue.1):52–61, August 2009.

[9] UC Davis Biometeorology Program - Frost Protection. http://biomet.ucdavis.edu/frost-protection.html.

[10] Valhalla Wireless. http://valhalla-wireless.com/vwr/.