DISHES: A Distributed Shell System Designed for Ubiquitous Computing Environment *

Chih-Chung Lai and Ren-Song Ko

National Chung Cheng University, Chia-Yi, Taiwan 621
{laicc,korenson}@cs.ccu.edu.tw

Abstract. This paper presents the design of the distributed shell system, DISHES, which serves as an interface middleware between a mobile user and ambient computers. With DISHES, a user can issue a command containing a sequence of programs and location of data to his mobile device, and the mobile device will retrieve the data from the specified location and automatically look for the required programs to process the data from the computing environment. Thus, a complicated task may be accomplished by gluing several primitive programs. These primitive programs may be executed on the more powerful computers nearby for better performance, and the results will be returned to the user. With this approach, the hardware and software of a mobile device may be kept as simple as possible; and thus the development costs may be reduced. We also propose a polynomial algorithm for finding a sequence of computers to execute the specified programs with the minimum network delay.

1 Introduction

Due to the advance of technology, various types of intelligent devices have pervaded into almost all faces of our daily life. We may have smart appliances, such as multimedia center, heating, ventilation, and air conditioning (HVAC), and or other white goods appliances, indoors while carrying mobile devices outsides. Examples of mobile devices are smart phones, PDAs, MP3 players, navigation systems, digital cameras, just to name a few. Besides, many public infrastructures, such as intelligent power, communication, and transportation systems, have been embedded with computers. In the future as we envisage it, networking these intelligent devices enables them to cooperate in support of better human living. Consequently, people are situated in the environment with many devices capable of computation and communication and exposed in networks; a computing environment called ubiquitous computing by Weiser [1].

However, the hardware constraints of many mobile devices often limit their usage. It is almost impossible to store huge amount of data and software in these devices as in PC. Furthermore, due to the limitation of battery power,

* This work is supported in part by the National Science Council under grants no. NSC 97-2218-E-194-003.

computation quality often degrades. For example, the AI and user interfaces of games on mobile devices are usually more primitive than those on PC. One approach to relieve the hardware constraint problem is to store data on other networked computers instead of being carried by users. When needed, a mobile user can download the data on demand via networks. To take this idea one step further, the computation resources needed to process the data may also be retrieved from networked computers. That is, we may download the required programs or software on demand to process the data or even ask other computers to execute the required programs, process the data and retrieve the results. Such an *on-site computing* approach relieves a mobile user's need to carry required programs and data on the road and, hence, boost people's mobility. By specifying the required programs and the location of data, mobile devices will automatically look for appropriate computers from computing environments to execute the required programs on the data retrieved from the specified location.

Based on the idea above, we propose a middleware, DISHES (DIstributed SHEll System), to mitigate the limitation caused by the hardware constraints of mobile devices. Users can issue a command specifying the data location and a sequence of program to process the data. When a mobile device receives the command, it will look for the appropriate computers nearby which have the required program and send the data location to the computers. Then these computers may download the data from the specified location and execute the required programs to process the data. Once finished, they will send the results to other computers for further processing or back to the original mobile device. Thus, a complicated task can be achieved by a sequential cooperation of multiple primitive programs.

The traditional shell programs on a single computer provide a good analogy of the idea above. In UNIX, execution of a program can be launched by issuing a command containing the program and the data to a shell program. Then the shell will look for the specified program among the default search paths and execute the program once found. Multiple programs can be combined to accomplish a complicated task via the pipe mechanism. Besides, the computation result can be redirected to a specific file via the I/O redirection mechanism. DISHES extends the shell on a single computer to the ubiquitous computing environment. With DISHES, a user can issue a command to a mobile device, and one of ambient computers having the required program will be "automatically" found. The computer will execute the required program on the data retrieved from the specified data location. DISHES also provides the remote pipe mechanism to enable a sequential cooperation of programs on different computers through networking. Similarly, after the computation is completed, the result can be sent back to the user's mobile device or stored in a specific location specified by the user via the remote I/O-redirection mechanism.

To illustrate our idea more explicitly, consider a scenario where there is a salesperson traveling to Chicago to visit his clients. To find out the visiting schedule in the order of clients' free time, he may issue the following command to his PDA:

```
grep Chicago http://mycompany.com/clients list | sort -k 2,2
> visit schedule
```

Based on the command, the salesperson's PDA will find the computers providing the required programs (grep and sort). Suppose computer A provides grep and computer B provides sort, as depicted in Fig.1. Then A will retrieve the client information from the specified data location, http://mycompany.com/clients list, execute grep Chicago to pick up the clients residing in Chicago, and send the intermediate result to B. After receiving the intermediate result from A, B will execute sort -k 2,2 to sort the clients in Chicago by their free time. Finally, the final sorted result will be sent back to the PDA and stored in the file, visit schedule.

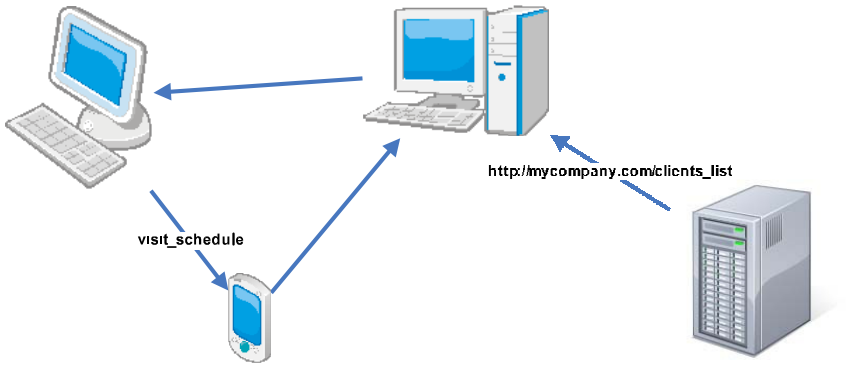

Fig. 1. The salesperson issues the command to his PDA to find the visiting schedule. Based on the command, the PDA will find the computers providing the specified programs, and the computers will execute the programs on the data retrieved from the specified URL.

In the above example, the salesperson only specifies the data location and a sequence of programs. DISHES will automatically look for appropriate computers with the required programs to process the data retrieved from the specified location. Moreover, a complicated task may be accomplished by gluing multiple primitive programs (grep and sort). These primitive programs do not have to be stored into and executed on the salesperson's PDA. They may be executed on the more appropriate computers nearby (A and B) for better performance, and the results will be returned to the user. With this approach, the hardware and software of a mobile device may be kept as simple as possible and, consequently, the device volume, weight, and cost can be minimized. Thus, DISHES boosts people's mobility. Besides, since a complicated task can be performed by gluing multiple primitive programs, the software development costs may be reduced.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. We describe the architecture of DISHES and explain the operations with the scenario above in the next section. In Section 4, we consider a performance optimization problem to minimize the network delay of DISHES. The problem can be formulated as the MINIMUM SEQUENTIAL WORK-

FLOW PROBLEM (MSWP) and we propose an algorithm with the complexity $O(m^2n^2)$, where n is the number of specified programs and n is the number of devices. Finally, we conclude this paper and discuss the future work with the current status of the project DISHES in the last two sections.

2 Related work

Shell programs, having a 30-years long developing history, provide a command line interface on many systems including UNIX which allows a user issue a command, parses the command into programs with parameters, and tell the system to execute the parsed programs. The first shell program, Bourne shell [2], often used in UNIX-like systems, was developed by Steve Bourne at Bell lab in 1974. In Bourne shell, a user can write scripts to assemble multiple primitive programs to form meaningful command for complicated computer tasks. Since then, several variations have been developed. Examples are C-shell [3] by Bill Joy in Berkeley, tcsh, Korn shell, and GNU Bash. They are different in script syntax and features such as job control, complex aliasing, I/O redirection, command line editing, filename completion, and interactive usage. GNU Bash, abbreviation of Bourne Again Shell, is one of the most popular shell programs in Linux. Bash is compatible with Bourne shell, and supports the convenient features of Korn shell and C-shell. Besides, it is compatible with IEEE POISX 1003.1 standard [4].

In recent years, various network technologies open up potential of many distributed applications and also push shell programs into new paradigms. Secure shell [5] provides a secure remote login mechanism by which a user can securely issue commands to control a remote computer. MOSIX [6] provides transparent execution of commands via a shell within a computer cluster consisting of workstations or PCs loosely coupled by network. Truong and Harwood [7] proposed a shell that provides distributed computing over a peer-to-peer network and is characterized by good scalability. What makes DISHES different than other shells is that DISHES aims to provide automatic program discovery functionality to utilize various types of ambient computers and help a mobile device to finish a computing task in a ubiquitous computing environment. Moreover, its remote file access functionality enables storing data and computation results in a distributed manner.

DISHES is inspired by the concept of ad hoc systems [8, 9]. An ad hoc system consists of multiple network connected computers or smart devices. To use these computers or devices to execute a resource intensive program, the program needs to be componentized and each component will be dispatched to the most suitable device determined by the underlying middleware for execution. Thus, a mobile device can utilize these ambient computers or devices to form an ad hoc system for a resource intensive program.

A related topic of combining multiple primitive programs for a complicated task is the composition of web services. The World Wide Web Consortium (W3C) defines the Web Services Description Language (WSDL) [10] to describe characteristics, such as input and output, of a web service. Programmers may use

WSDL to specify how to access web services, and thus web services can be composed to provide a complicated service. However, WSDL does not support semantics description; thus, a composition always requires programmers' intervention.

To enable web services to perform a dynamic composition by themselves, Martin et al. proposed the OWL-S approach [11]. With OWL-S, a client program and web services may have a common consensus of the semantics of the terms in WSDL by an ontology on a third-party, and thus a web service can automatically interact with another web service by a priori setting the rule for the semantics. Based on the OWL-S approach, Mokhtar et al. proposed a conversation-based service composition method, COCOA [12], that aims for dynamic composition of services to complete a user task. With COCOA, a service as well as a user task is transformed to an automata, and an algorithm is proposed to combine the automata of services.

QoS-aware composition is another important issue of web service composition. The objective is to compose web services to satisfy given optimization criteria such as the overall cost or response time, and can be formulated as a NP-hard optimization problem [13]. Canfora et al. [13] proposed a genetic algorithm for the NP-hard QoS-aware composition problem. Besides, Wada et al. [14] proposed a multi-objective genetic algorithm to deal with optimization criteria with trade-off, and Berbner et al. [15] proposed a fast heuristic that has 99% close to optimal solutions in most cases. Furthermore, various middleware and frameworks are proposed to realize QoS-aware web service composition [16–18].

3 Architecture of DISHES

DISHES consists of five subsystems, namely, *Service Discovery (SD)*, *Admin (AD)*, *User Interface (UI)*, *Command Dealer (CD)*, and *Pipe-I/O-Exec (PIE)*. Fig.2 depicts an overview of the system architecture of DISHES and each subsystem is briefly described in the following subsections.

3.1 Service Discovery (SD) subsystem

The main task of *SD* is to provide the functionality of automatic program discovery. *SD* works as a broker – it provides registration and lookup service of a program via two types of requests, the **registration** request and lookup request, from other subsystems.

When a computer wants to provide a service of executing a specific program, it may use the **registration** request to state its intention. The **registration** request contains all the necessary attributes for discovery such as the program name and the computer's network address. When receiving the **registration** request, *SD* stores the program name and the host computer's network address into the database, and replies with the **success** or **fail** message.

When a device wants to know which computers provide a specific program, it may query *SD* by using the lookup request containing the program name. On

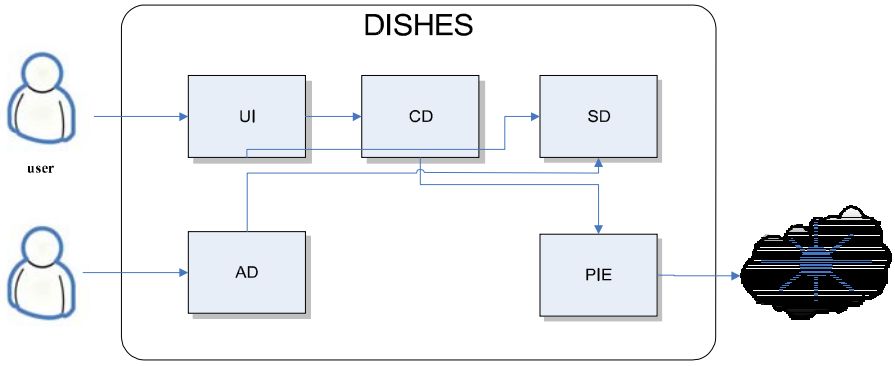

Fig. 2. An overview of the system architecture of DISHES: an arrow represents an interaction existing between two components and the direction indicates that the workflow is from the sending subsystem to the receiving subsystem.

receiving the lookup request, *SD* will search the program name in the database and reply the device with the corresponding information of all the host computers that have registered the program.

3.2 Admin (AD) subsystem

AD receives configuration information from the system administrator, and sends the registration requests to *SD* to register the programs and the corresponding host computer information.

3.3 User Interface (UI) subsystem

A user can issue a command or a script of commands to *UI*. Then, *UI* will parse and interpret the command or script. When a specified program is absence in the local device, *UI* will send a lookup request to *SD* to retrieve the information of the hosts of the specified program. After retrieving the information of all the hosts, *UI* will marshal the command into a standardized format, including the program name concatenated with the host network address and the data file path concatenated with the network address of the data storage. The purpose of the marshalling process is to provide sufficient information where the program or the data file can be located for the subsequent execution of the command. After that, *UI* will send the marshalled command to *CD*.

UI usually resides in a user's mobile device and *SD* resides in a centralized server. When the mobile device moves to a new environment, the *UI* on the mobile device may find a *SD* through broadcasting to the local network.

3.4 Pipe-I/O-Exec (PIE) subsystem

Execution of a marshalled command basically involves a sequence of interactions between *CD* and *PIE*, where *CD* parses the marshalled command and asks *PIE*

to perform specified tasks. This subsection describes the services provided by *PIE*, and the next section describes how *CD* asks *PIE* to handle a marshalled command.

PIE as the name indicates is responsible for three types of services, namely, remote pipe, remote I/O redirection, and remote execution. The remote pipe request is used to construct a virtual pipe from the standard output of a process to the standard input of a remote process through the network. Such a remote pipe can be realized by the assistance of two extra processes as depicted in Fig. 3. For example, in order to construct a remote pipe from process P_A of computer A to process P_B of computer B, two assistant processes, AP_A on A and AP_B on B, are created with two regular UNIX pipes, P_A to AP_A and AP_B to P_B. Moreover, a socket connection between AP_A and AP_B is constructed, and the standard output of AP_A is redirected to one end of the socket and the other end of the socket is redirected to the standard input of AP_B. Consequently, we have a remote pipe from P_A to P_B, in which the information actually flows from P_A to AP_A via one regular UNIX pipe, then AP_A to AP_B via a socket connection, and finally AP_B to P_B via the other regular UNIX pipe.

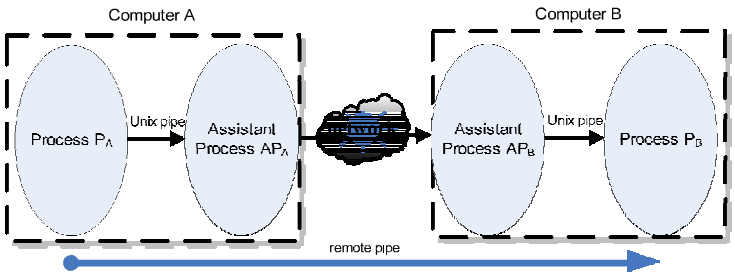

Fig. 3. The diagram of constructing a remote pipe

The remote I/O redirection request is used to redirect the standard output of a process to a remote file through the network. Similar to the remote pipe, two assistant processes, AP_A on A and AP_B on B, are required as illustrated in Fig. 4. The output of P_A is redirected to AP_A, and then to AP_B via a socket connection. Finally, the output is saved to file F_B via the regular I/O redirection mechanism.

The remote execution request is used to execute a program on a remote host computer. If the program is given an URL indicating the data file location, *PIE* on the host will retrieve the file by the URL, and store the file for input data before execution.

3.5 Command Dealer (CD) subsystem

CD processes a marshalled command in three phases, namely, pipe phase, I/O redirection phase, and execution phase.

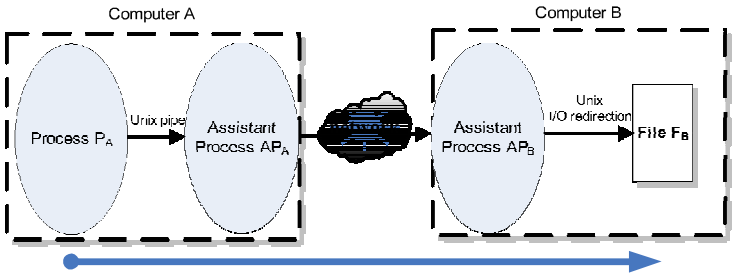

Fig. 4. The diagram of constructing a remote I/O redirection

The pipe phase is to create necessary remote pipes. When a pipe token, "|", is parsed from a command, *CD* will send a remote pipe request with the network addresses of the source computer and the destination computer to *PIE* for constructing a remote pipe. If a computer has already created an assistant process, the assistant process will be reused.

The I/O redirection phase is to create necessary remote I/O redirections. When an I/O redirection token, ">", is parsed from a command. *CD* will send a remote I/O redirection request with the network addresses of the source computer and the destination computer and the file name to *PIE* for constructing a remote I/O redirection. If a computer has already created an assistant process, the assistant process will be reused. Otherwise, a new assistant process will be created for use.

The execution phase is to perform remote executions. When a program name is parsed from a command, *CD* will send a remote execution request with the network addresses of the host and the parameters of the program to *PIE* for executing the program on the host. If the host has already created a process for execution, the process is reused. Otherwise, a new process will be created for executing the program. The order of remote program executions is the reverse of the order that the programs appear in the marshalled command. The reason is that the receiving process of a remote pipe must execute the specified program before the sending process outputs result to the remote pipe or the result may be lost.

Fig.5 is an example of how *CD* processes the marshalled command,

progA@addressA | progB@addressB > FC@addressC

In the pipe phase, *CD* sends a remote pipe request with addressA and addressB to *PIE*, and then process P_A and P_B are created and connected by a remote pipe. In the I/O redirection phase, since the device in addressB has created P_B during the previous phase, P_B will be also used as the source process to remotely redirect I/O to file F_C on the computer at addressC. In the execution phase, progB is executed on the computer at addressB first, and then progA is executed on the computer at addressA. After the execution is finished, the final output is

stored as file F_C on the device at addressC. Note that we ignore the details, such as assistant processes, regular pipes, regular I/O redirection, and the sockets required for the remote pipe and I/O redirection in the figure.

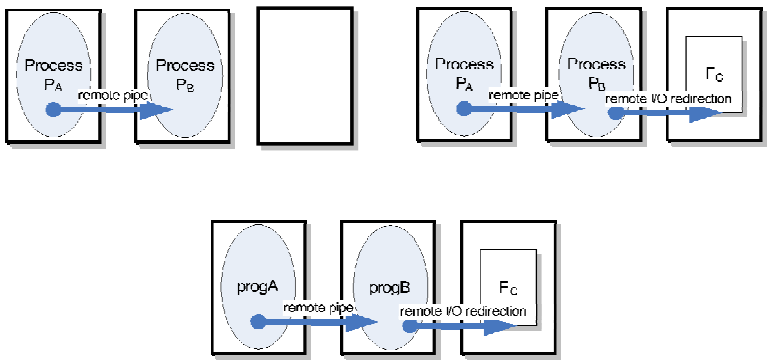

Fig. 5. The three phases that *CD* processes the marshalled command " progA@addressA | progB@addressB > FC@addressC". Note in the execution phase, progB is executed before progA.

3.6 An operational example

Fig. 6 illustrates the operations of DISHES for the scenario mentioned in Section 1. The operations basically occur in the following sequences.

1. The *Admin* on computer B send a **registration** request with **sort** to the *SD* on a nearby server.
2. The *Admin* on computer A send a **registration** request with grep to the *SD* on a nearby server.
3. The *UI* on the PDA receives the command,

   grep Chicago http://mycompany.com/clients **list** | sort -k 2,2 > **visit** schedule

   issued by the salesperson.
4. The *UI* on the PDA finds that there is no grep or **sort** on the PDA, and sends lookup requests for grep and **sort** to the *SD* on the nearby server.
5. The *SD* on the nearby server replies the information of hosts, A and B, to the *UI* on the PDA.
6. The *UI* on the PDA marshals the command by attaching the network address of A and B to grep and **sort**. Denote the address of A as adr(A), the network address of B as adr(B), and the network address of the PDA as adr(PDA). Hence, the input command will be marshalled as

grep@adr(A) Chicago http://mycompany.com/clients list | sort@adr(B) -k 2,2 > visit schedule@adr(PDA)

and the marshalled command will be sent to the *CD* on the PDA.

7. In the pipe phase, the *CD* on the PDA sends a remote pipe request to the *PIE* on the PDA to notify the *PIE* on A to construct a remote pipe to B.
8. In the I/O redirection phase, the *CD* on the PDA sends a remote I/O redirection request to the *PIE* on the PDA to notify the *PIE* on B to construct a remote I/O redirection to file, visit schedule, on the PDA.
9. In the execution phase, the CD sends a remote execution request with "sort@adr(B) -k 2,2" to the *PIE* on the PDA to notify the *PIE* on B to execute "sort -k 2,2". Then *PIE* on B starts to execute "sort -k 2,2".
10. Next, the *CD* sends a remote execution request with "grep@adr(A) Chicago http://mycompany.com/clients list" to the *PIE* on the PDA to notify the *PIE* on A to execute "grep Chicago http://mycompany.com/clients list".
11. The *PIE* on A finds there is a input data file specified by "http://mycompany.com/clients list", so it will retrieve the file by the URL and store the file as clients list in the local working directory of A. Then the *PIE* on A starts to execute "grep Chicago clients list".

After the above operations, A will execute "grep Chicago clients list", and send the intermediate result to B through the remote pipe constructed in operation 7. Meanwhile, B has executed "sort -k 2,2", which is waiting for the intermediate result from A as input. Finally, B will send the sorted result to file visit schedule on the PDA through the remote I/O redirection constructed in operation 8. Fig. 7 depicts the interactions among the programs, the remote pipe, and the remote I/O redirection.

4 Performance optimization

Note that the programs specified by a command may be sequentially executed on different computers. The intermediate execution results of one computer may need to be transferred to another computer for subsequent program execution via the network. Because of the communication time needed for the intermediate execution results, to find a sequence of devices so that the total communication time is minimum is an important task of DISHES for performance optimization.

Consider the scenario where "P_1 | P_2 | P_3" is the command for execution. P_1 is available(can be executed) on computers v_1, v_2, and v_3, P_2 is available on v_2, v_4, and v_5, and P_3 is availabe on v_1 and v_6. Suppose the netwrok delays between any pair of devices are known. The goal is to find a sequence of three devices $(v_{(1)}, v_{(2)}, v_{(3)})$ such that P_1, P_2, and P_3 are executed on $v_{(1)}$, $v_{(2)}$, and $v_{(3)}$ respectively and the total network delay $d(v_{(1)}, v_{(2)}) + d(v_{(2)}, v_{(3)})$ is minimized, in which $d(u, v)$ is the network delay from u to v.

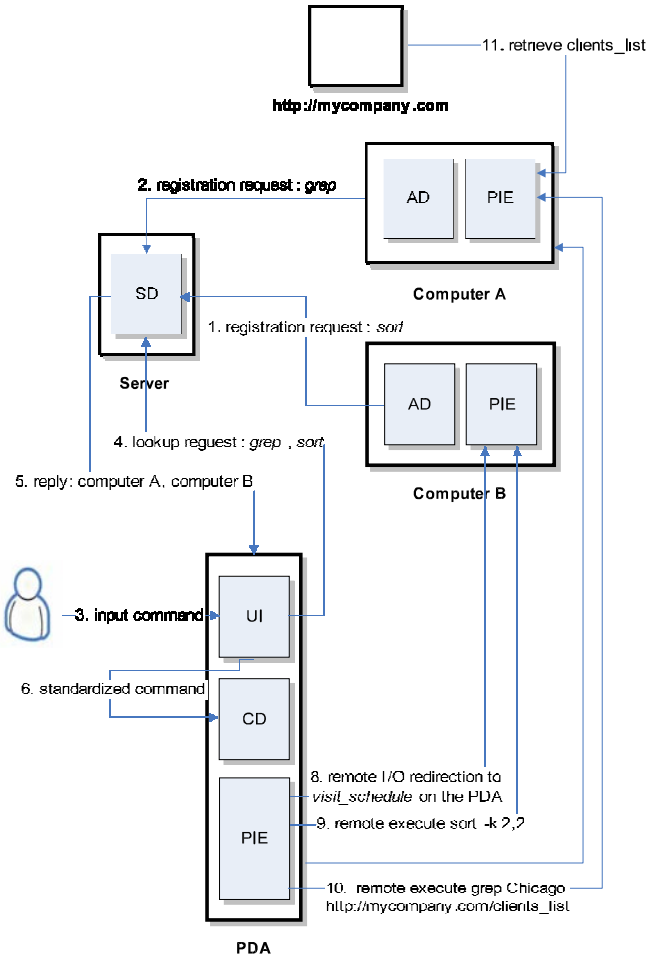

Fig. 6. An operational example of DISHES

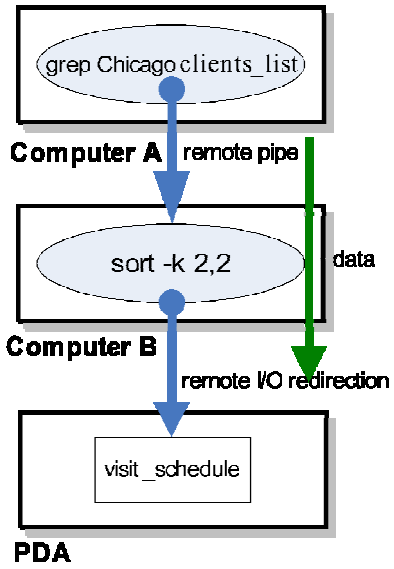

Fig. 7. The data flows through remote pipes and remote I/O redirection.

Thus, we may formalize the performance optimization problem as follows and derive some facts.

Definition 1. *Given a set of computers* V, *a sequence of programs* $S_P = (p_{(1)}, p_{(2)}, \ldots, p_{(m)})$, *and a set of host computers* $A(p_{(i)}) \subseteq V$ *for each* $p_{(i)} \in S_P$. *A sequence of computers* $S_V = (v_{(1)}, v_{(2)}, \ldots, v_{(m)})$, *is* feasible *if* $v_{(i)} \in A(p_{(i)})$ *for* $1 \le i \le m$.

Problem 1. MINIMUM SEQUENTIAL WORKFLOW PROBLEM (MSWP) Given a set of computers V, a sequence of programs $S_P = (p_{(1)}, p_{(2)}, \ldots, p_{(m)})$, a set of host computers $A(p_{(i)}) \subseteq V$ for each $p_{(i)} \in S_P$, and delay function $d(v_i, v_j)$ for all $v_i, v_j \in V$. Find a feasible sequence of computers $S_V = (v_{(1)}, v_{(2)}, \ldots, v_{(m)})$, such that the total delay $\sum_{i=1}^{m-1} d(v_{(i)}, v_{(i+1)})$ is minimum.

Definition 2. *Given an instance* I *of MSWP:* V, S_P, $A(p_{(i)})$ *for each* $p_{(i)} \in S_P$, *and* $d(v_i, v_j)$ *for all* $v_i, v_j \in V$. *A sequential workflow induced graph (SWIG) of* I *is a directed graph* $G_I(V_I, E_I)$ *with a cost function* $c_I(e)$ *for all* $e \in E_I$. *Here*

$$V_I = \{v_i^j / v_i \in A(p_{(j)}) \text{ for } 1 \le j \le m\} \cup \{v_s, v_e\},$$
$$E_I = E_s \cup E_I^i \cup E_e, \text{ in which}$$
$$E_s = \{(v_s, v_i^1) / v_i \in A(p_{(1)})\},$$
$$E_I^i = \{(v_i^k, v_j^{k+1}) / v_i \in A(p_{(k)}) \text{ and } v_j \in A(p_{(k+1)}) \text{ for } 1 \le k \le m-1\},$$
$$E_e = \{(v_i^m, v_e) / v_i \in A(P_{(m)})\},$$
$$c_I(e) = \begin{cases} d(v_i, v_j), & \text{if } e = (v_i^k, v_j^{k+1}) \in E_I^i \\ 0, & \text{if } e \in E_s \cup E_e \end{cases}$$

Lemma 1. *A feasible sequence of an instance* $\boldsymbol{I}$ *can be derived from a path from* v_s *to* v_e *of the corresponding SWIG* $G_{\boldsymbol{I}}$. *Besides, a path from* v_s *to* v_e *of* $G_{\boldsymbol{I}}$ *can be derived from a feasible sequence for* $\boldsymbol{I}$.

Proof. A path from v_s to v_e of $G_{\boldsymbol{I}}$ must be in the form of $(v_s, v^1_{k_{(1)}}, v^2_{k_{(2)}}, \ldots, v^m_{k_{(m)}}, v_e)$, so $\langle v_{k_{(1)}}, v_{k_{(2)}}, \ldots, v_{k_{(m)}} \rangle$ is a feasible sequence by Definition 2. On the other hand, by Definition 2 and 1, for a feasible sequence $\langle v_{k_{(1)}}, v_{k_{(2)}}, \ldots, v_{k_{(m)}} \rangle$, $(v_s, v^1_{k_{(1)}}, v^2_{k_{(2)}}, \ldots, v^m_{k_{(m)}}, v_e)$ is a path from v_s to v_e of $G_{\boldsymbol{I}}$.

From Lemma 1, we may call $\langle v_{k_{(1)}}, v_{k_{(2)}}, \ldots, v_{k_{(m)}} \rangle$ the counterpart sequence of the path $(v_s, v^1_{k_{(1)}}, v^2_{k_{(2)}}, \ldots, v^m_{k_{(m)}}, v_e)$ and $(v_s, v^1_{k_{(1)}}, v^2_{k_{(2)}}, \ldots, v^m_{k_{(m)}}, v_e)$ the counterpart path of $\langle v_{k_{(1)}}, v_{k_{(2)}}, \ldots, v_{k_{(m)}} \rangle$.

Definition 3. *The cost of a path* $(v_{(1)}, v_{(2)}, \ldots, v_{(l)})$ *in a SWIG is defined as*

$$\sum_{i=1}^{l-1} c_{\boldsymbol{I}}((v_{(i)}, v_{(i+1)}))$$

Lemma 2. *The cost of a path from* v_s *to* v_e *equals the total delay of its counterpart feasible sequence, and the total delay of a feasible sequence equals the cost of its counterpart path.*

Proof. By Definition 2 and 3, the cost of path $(v_s, v^1_{k_{(1)}}, v^2_{k_{(2)}}, \ldots, v^m_{k_{(m)}}, v_e)$ is

$$\sum_{i=1}^{m-1} c_{\boldsymbol{I}}((v^i_{k_{(i)}}, v^{i+1}_{k_{(i+1)}})) + c_{\boldsymbol{I}}((v_s, v^1_{k_{(1)}})) + c_{\boldsymbol{I}}((v^m_{k_{(m)}}, v_e)) = \sum_{i=1}^{m-1} c_{\boldsymbol{I}}((v^i_{k_{(i)}}, v^{i+1}_{k_{(i+1)}})) =$$

$$\sum_{i=1}^{m-1} d(v_{k_{(i)}}, v_{k_{(i+1)}}) = \text{total delay of } \langle v_{k_{(1)}}, v_{k_{(2)}}, \ldots, v_{k_{(m)}} \rangle.$$

Theorem 1. *The counterpart sequence of the shortest path, i.e., the one with the minimum cost, from* v_s *to* v_e *of* $G_{\boldsymbol{I}}$ *is the feasible sequence with the minimum total delay for a MSWP instance* $\boldsymbol{I}$.

Proof. We prove this theorem by contradiction. Suppose there exists a feasible sequence S_V having less total delay than the counterpart sequence of the shortest path denoted as PA_{opt}. By Lemma 1 and 2, the counterpart path of S_V has less cost than PA_{opt}, which leads to a contradiction.

Based on Theorem 1, we propose an algorithm for MSWP illustrated in Algorithm 2. Note that Algorithm 2 basically contains two parts, Algorithm 1 for the construction of $G_{\boldsymbol{I}}$ (line 1) and Dijkstra's algorithm for the shortest path (line 2). Refer to Algorithm 1 and Definition 2, the time complexity to construct $V_{\boldsymbol{I}}$ (line 3-9) is $O(\sum_{i=1}^{m} |A(p_{(i)})|)$ and to construct $E_{\boldsymbol{I}}$ and $c_{\boldsymbol{I}}$ (line 10-25) is $O(\sum_{i=1}^{m-1} |Ap_{(i)}| \times |A(p_{(i+1)})|) + O(|A(p_{(1)})|) + O(|A(p_{(m)})|)$, in which each term

Algorithm 1 Construction of SWIG

Input: an instance I of MSWP: V, S_P, $A(p_{(i)})$ for each $p_{(i)} \in S_P$, $d(v_i, v_j)$ for all $v_i, v_j \in V$,

Output: a SWIG $G_I(V_I, E_I)$ with cost funtion c_I

1: $V_I \leftarrow \emptyset$
2: $E_I \leftarrow \emptyset$
3: for $j = 1$ to m do
4: for all $v_i \in A(p_j)$ do
5: $V_I \leftarrow V_I \cup \mathrm{v}_i^j$
6: end for
7: end for
8: $V_I \leftarrow V_I \cup \mathrm{v}_s$
9: $V_I \leftarrow V_I \cup \mathrm{v}_e$
10: for $k = 1$ to $m - 1$ do
11: for all $v_i \in A(p_k)$ do
12: for all $v_j \in A(p_{k+1})$ do
13: $E_I \leftarrow E_I \cup (\mathrm{v}_i^k, \mathrm{v}_j^{k+1})$
14: $c_I((\mathrm{v}_i^k, \mathrm{v}_j^{k+1})) = d(v_i, v_j)$
15: end for
16: end for
17: end for
18: for all $v_i \in A(p_1)$ do
19: $E_I \leftarrow E_I \cup (\mathrm{v}_s, \mathrm{v}_i^1)$
20: $c_I((\mathrm{v}_s, \mathrm{v}_i^1)) = 0$
21: end for
22: for all $v_i \in A(p_m)$ do
23: $E_I \leftarrow E_I \cup (\mathrm{v}_i^m, \mathrm{v}_e)$
24: $c_I((\mathrm{v}_i^m, \mathrm{v}_e)) = 0$
25: end for
26: return $G_I(V_I, E_I)$ with c_I

Algorithm 2 Solution of MSWP

Input: an instance I of MSWP: V, S_P, $A(p_{(i)})$ for each $p_{(i)} \in S_P$, $d(v_i, v_j)$ for all $v_i, v_j \in V$,

Output: a sequence of devices S_V with minimum $\sum_{i=1}^{m-1} d(v_i, v_{i+1})$

1: Construct a SWIG G_I from I by Algorithm 1.
2: Apply Dijkstra's algorithm [19] on $G_I(V_I, E_I)$ to find a path $(\mathrm{v}_s, \mathrm{v}^1_{opt_{(1)}}, \mathrm{v}^2_{opt_{(2)}}, \ldots, \mathrm{v}^m_{opt_{(m)}}, \mathrm{v}_e)$ from v_s to v_e with minimum cost.
3: Derive the counterpart sequence $(v_{opt_{(1)}}, v_{opt_{(2)}}, \ldots, v_{opt_{(m)}})$ from the path found in the previous step.
4: $S_V \leftarrow (v_{opt_{(1)}}, v_{opt_{(2)}}, \ldots, v_{opt_{(m)}})$

accounts for the complexity for constructing E_I^i (line 10-17), E_s (line 18-21), E_e (line 22-25) respectively.. Thus, for the construction of G_I, the total time complexity is $O(\sum_{i=1}^{m} |A(p_{(i)})|) + O(\sum_{i=1}^{m-1} |Ap_{(i)}| \times |A(p_{(i+1)})|) + O(|A(p_{(1)})|) + O(|A(p_{(m)})|) = O(\sum_{i=1}^{m-1} |Ap_{(i)}| \times |A(p_{(i+1)})|)$. Besides, the time complexity of Dijkstra's algorithm is $O((\sum_{i=1}^{m} |A(p_{(i)})|)^2)$. Therefore the total complexity of Algorithm 2 is $O(\sum_{i=1}^{m-1} |Ap_{(i)}| \times |A(p_{(i+1)})| + \sum_{i=1}^{m} |A(p_{(i)})|)^2) = O((\sum_{i=1}^{m} |A(p_{(i)})|)^2) = O(m^2n^2)$, where m is the number of specified programs and n is the number of computers.

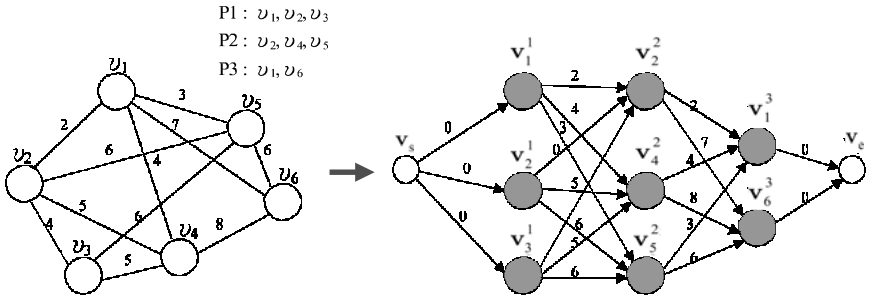

Fig. 8. An example instance of MSWP mentioned earlier in this section is transformed to SWIG by Algorithm 1. The left side is the graph representation of the example instance with the network delays between computers and the right side is the derived SWIG. Vertex v_i^j of the SWIG indicates that the computer v_i is one host of program P_j, e.g., $v^1_{\ 2}$ for $v_2 \in A(P_2)$, and the cost of edge (v^j_i, v^{j+1}_k) is equal to the network delay between computers v_i and v_k, e.g., the cost of edge (v_1^1, v_2^2) is $d(v_1, v_2) = 2$. The costs of edges from v_s to the vertices with superscript 1 and edges from the vertices with superscript 3 to v_e are 0. After SWIG is derived, the shortest path from v_s to v_e can be determined by Dijkstra's algorithm, i.e., $(v_s, v^1_{\ 2}, v_2^2, v_1^3, v_e)$, and its counterpart sequence (v_2, v_2, v_1) is the feasible sequence with the minimum total delay.

Figure. 8 illustrates the application of Algorithm 1 on the example instance desribed earlier in this section. The figure on the left side is the graph representation of the example instance which depicts the network delays between computers. The figure on the right side is the SWIG converted from the example instance by Algorithm 1. After applying Dijkstra's algorithm on the SWIG, the shortest path $(v_s, v^1_{\ 2}, v_2^2, v_1^3, v_e)$ will be found, and the counterpart sequence (v_2, v_2, v_1) is the feasible sequence with the minimum total delay.

Note that it is easy to extend MSWP and SWIG to model the execution of the commands with remote I/O redirection and with remote file retrieval, and hence the corresponding performance optimization problem can be solved

by minor modification on Algorithm 2. Remote I/O redirection basically is an operation on the redirection destination computer which redirects I/O to the redirected file. Thus the operation can be treated as a specified program in MSWP and its only host is the redirection destination computer. For example, for the command, " F_{input}@address(v_1) > P_1 | P_2 > F_{output}@address(v_2)", S_P is $(OP_{input}, P_1, P_2, OP_{output})$ and $A(OP_{input}) = \{v_1\}$ and $A(OP_{output}) = \{v_2\}$, in which OP_{input} and OP_{output} are the operations redirecting I/O to F_{input} and F_{output} respectively. Moreover, for the program execution on a remote data file, the network delay from the remote file location to each host of the specified program should be included in the cost of each edge starting from the host to account for the network delay of remote file retrieval. For example, for the command "P_1 | P_2 http://address(v_1)/F_{data} | P_3", each host of P_2 needs to download file F_{data} from v_1 before the execution of P_2, so $d(v_1, v_i)$, for all $v_i \in A(P_2)$, must be added to the network delay between v^2_i and v^3_j for the cost of (v^2_i, v^3_j) for all $v_j \in A(P_3)$, i.e., $c_I((v^2_i, v^3_j)) = d(v_i, v_j) + d(v_1, v_i)$.

5 Conclusion and future work

The realization of ubiquitous computing brings mobile users new computing experiences. Mobile users can be served by various types of ambient computers with mobile devices playing the role of the bridges between users and ambient computers. Thus, there is a need for middleware that can bridge users and their computing environments via mobile devices.

To fit the necessity, this paper proposes the design of a distributed shell system, DISHES. DISHES provides remote pipe, remote I/O redirection, remote execution, remote file retrieval, and automatic program discovery. With these functionalities and pervasiveness of computers, a mobile device can easily find and use programs provided by the computers nearby to process the data on network, and results can be retrieved or stored in a specified location. With this approach, the hardware and software of a mobile device may be kept as simple as possible and, consequently, the device volume, weight, and cost can be minimized. Thus, DISHES boosts people's mobility. Besides, a complicated task can be accomplished by gluing multiple primitive programs, so the software development costs may be reduced.

Since the intermediate execution results of one computer may need to be transferred to another computer for subsequent program execution via the network, we also consider the problem to find a sequence of computers so that the total communication time is minimum. Such a performance optimization problem can be formulated as MSWP and we propose an algorithm with the quadratic asymptotic complexity in terms of the number of specified programs and computers.

Service discovery in the current implementation is primitive. It lacks of considering several important issues in ubiquitous computing such as semantics matching [20], load balancing, or security [21]. In the future, we shall investigate these issues for more sophisticated service discovery design. Furthermore,

a priori assumption of the performance optimization problem considered is that the computing environment is static. Thus, DISHES has the knowledge of all ambient computers, a prerequisite of the proposed algorithm. However, this assumption may not reflect the dynamic characteristic of a ubiquitous computing environment where computers may join or leave at their wills. Besides, in some cases, users may enter an unfamiliar computing environment and do not have any knowledge of available computing resources. We shall extend the performance optimization problem to on-line version and propose a distributed algorithm with guaranteed performance.

6 Status of DISHES

DISHES is an open-source software project [22]. The development process follows the lightweight CMMI. At the time of writing this paper, the project is at the stage of final integration. The implementation is expected to be completed in October, 2009 and to be released under GNU General Public License 2.0 (GPLv2).

Acknowledgement

Thanks to all other members of project DISHES, Chia-Kuan Yen, Po-Liang Lin, Kai-Wen Shih, and Shan Kuan for contributing their ideas and effort. Also especially thanks to National Science Council and Academia Sinica for supporting this project.

References

1. Weiser, M.: The Computer for the Twenty-First Century. Scientific American (September 1991) 94–104
2. Bourne, S.R.: The UNIX shell. The Bell System Technical Journal 57(6(part 2)) (1971)
3. : TCSH(1), Astron 6.10.01 (26 April 2001), information available at http://www.tcsh.org/tcsh.html/top.html.
4. : IEEE Std 1003.1, 2003 Edition, information available at http://www.unix-systems.org/version3/ieee std.html.
5. : OpenSSH, information available at http://www.openssh.com/.
6. Barak, A., Wheeler, R.: MOSIX: An Integrated Multiprocessor UNIX. In: USENIX Technical Conference Proceedings. (1989)
7. Truong, M., Harwood, A.: Distributed shell over peer-to-peer networks. In: In Proceeding of the International Conference on Parallel and Distributed Processing Techniques and Applications, pages 269V275, Las Vegas, NV. (2003) 269–275
8. Ko, R.S.: ASAP for Developing Adaptive Software within Dynamic Heterogeneous Environments. phd thesis, department of computer science and engineering, michigan state university (May 2003)

9. Ko, R.S., Lai, C.C., Yen, C.K., Mutka, M.W.: Component-Based Ad Hoc Systems for Ubiquitous Computing. International Journal of Pervasive Computing and Communications 4(4) (2008)
10. : Web Services Description Language (WSDL) 1.1, 2001, information available at http://www.w3.org/TR/wsdl.
11. Martin, D., Paolucci, M., Mcilraith, S., Burstein, M., Mcdermott, D., Mcguinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing Semantics to Web Services: The OWL-S Approach, Springer (2004) 26–42
12. Mokhtar, S., Georgantas, N., Issarny, V.: COCOA : ConversationBased Service Composition for Pervasive Computing Environments. International Conference on Pervasive Services 0 (2006) 29–38
13. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An Approach for QoS-Aware Service Composition Based on Genetic Algorithms. In: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM (2005) 1069–1075
14. Wada, H., Champrasert, P., Suzuki, J., Oba, K.: Multiobjective Optimization of SLA-Aware Service Composition. Services, IEEE Congress on 0 (2008) 368–375
15. Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware Web Service Composition. Web Services, IEEE International Conference on 0 (2006) 72–82
16. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Transactions on Software Engineering 30(5) (2004) 311–327
17. Issa, H., Assi, C., Debbabi, M.: QoS-Aware Middleware for Web Services Composition - A Qualitative Approach. Computers and Communications, IEEE Symposium on 0 (2006) 359–364
18. Yu, T., Lin, K.J.: A Broker-Based Framework for QoS-Aware Web Service Composition. e-Technology, e-Commerce, and e-Services, IEEE International Conference on 0 (2005) 22–29
19. Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. Numerische Mathematik 1 (1959) 269–271
20. Lord, P., Alper, P., Wroe, C., Goble, C.: Feta: A Light-Weight Architecture for User Oriented Semantic Service Discovery. Lecture Notes in Computer Science 3532 (2005) 17–31
21. Czerwinski, S.E., Zhao, B.Y., Hodes, T.D., Joseph, A.D., Katz, R.H.: An Architecture for A Secure Service Discovery Service. In: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking, ACM (1999) 26–35
22. : OpenFoundry Project, DISHES, information available at http://of.openfoundry.org/projects/1052.