Dependency Management In Distributed Settings

Guanling Chen and David Kotz
Department of Computer Science, Dartmouth College
Hanover, NH 03755, USA
{glchen, dfk}@cs.dartmouth.edu

Dartmouth Computer Science Technical Report TR2004-495

Abstract

Ubiquitous-computing environments are heterogeneous and volatile in nature. Systems that support ubicomp applications must be self-managed, to reduce human intervention. In this paper, we present a general service that helps distributed software components to manage their dependencies. Our service proactively monitors the liveness of components and recovers them according to supplied policies. Our service also tracks the state of components, on behalf of their dependents, and may automatically select components for the dependent to use based on evaluations of customized functions. We believe that our approach is flexible and abstracts away many of the complexities encountered in ubicomp environments. In particular, we show how we applied the service to manage dependencies of context-fusion operators and present some experimental results.

1 Introduction

Many ubiquitous-computing (ubicomp) applications are designed to reduce complexity in their users' lives by adapting to the context around the user. These environments include a wide variety of mobile devices, often connected by wireless networks. As a result, the set of devices, users, and conditions is highly dynamic. To limit the complexity faced by ubicomp application programmers, ubicomp system software should be self-organized and self-repaired to require minimum human intervention.

One aspect of coordination is to manage the dependencies among software components, particularly when they are temporally coupled [3]. We say a component X *depends* on another component Y if X needs service from Y, and we say X is a *dependent* of Y. The problem of dependency management, in face of potentially unexpected failures, has two aspects: one is to monitor the component's liveness and restart it in time, and the other is to control the components according to dependency *policies*. There are many possible policies, as we discuss in detail below; they may include rules such as reclaiming a component when it has no dependents, or selecting a different component when the original one failed.

In this paper we present a general infrastructure that monitors and recovers distributed services, which may fail, migrate, or disconnect during hand-offs. The system provides a common name space that allows service discovery and composition [16]. It also executes component objects supplied by applications for customization, tailed to their specific needs [19]. These component objects may also be lost due to host crashes, and may migrate for purpose of balancing CPU load and network usage. The resulting mobility requires the system to track component locations to facilitate communications between components and their dependents.

As a result, a ubicomp support infrastructure should include 1) a method for a service to register a name and information about how to restart itself in case of failure, 2) a method for applications to query the name space to locate available services, 3) a way for components to register dependency relationships and associated policies, 4) a method for applications to inject additional objects into the data-flow path from services to themselves, and 5) a method for communication between components and their dependents. By abstracting most of the complexity of these functions, developing a reliable ubicomp applications becomes a much easier task.

We assume that each component is given a globally unique numeric key that is invariant once the component is registered, even if it is restarted or moved to another host later. Thus, we can specify the dependency relationship between two components as if one key depends on another, for example, $K_X \rightarrow K_Y$ if X depends on Y and K_X and K_Y are their keys. We also assume that the network link failures do not partition the network, and that failing components simply stop and disappear (no Byzantine faults). To simplify the design, we also consider the components are trusted.

In this paper, we present a specific ubicomp system, *Solar*, that provides a collection of services for ubicomp applications. In particular, Solar provides a *context fusion* service that allows applications to compose context information by subscribing to events from a set of sensors and other information sources. Solar applications insert additional *operators* to aggregate low-level data into high-level knowledge. Autonomic systems may make adaptation decisions, for example, by using the Solar service as a substrate to infer current the state of the devices, network, and users.

Solar consists a set of infrastructure nodes, which we call *Planets*, peering together to form a service overlay built with a self-organized and self-repaired peer-to-peer routing protocol [2]. Each Planet is an overlay node and has its own unique numeric key. The Planets cooperatively execute the operators, which are software components with their own keys. Solar may move operators at run time to balance the load on Planets. An operator takes one or more event streams as input and produces an event stream, so operators can be stacked together to form a directed graph. The dependencies among data sources, operators, and applications are monitored and managed by Solar's *dependency* service.

The rest of the paper is organized as follows. We present background information about Solar in Section 2. Section 3 discusses design details of Solar's dependency service, and Section 4 contains some implementation issues. We give experimental results in Section 5. Section 6 discusses related work and we conclude in Section 7.

2 Background

Ubiquitous-computing applications must adapt to various computational, environmental, and user state to reduce the need for user intervention [18]. This sort of information is *context*, that is, information that affects how an application performs its task. Context is typically a piece of high-level knowledge that may be derived from distributed sensors producing potentially low-quality data. For instance, the content delivered to a user may be customized by the user's current location, which might be aggregated from several data sources [13]. These context-aware applications need system support for collection, aggregation, and dissemination of contextual information. The supporting system itself, however, should be autonomic, self-organized, and self-repaired.

2.1 Context fusion

Solar provides a flexible and scalable infrastructure to connect data sources, and allows applications to aggregate and customize contextual information from sources by injecting data-fusion operators. The operators could be simple filters or they may employ more sophisticated machine-learning algorithms. An operator takes input from one or more data sources and acts as another source producing data. The simple interface allows operators be stacked together to form an acyclic *operator graph* for more complex context computation [8].

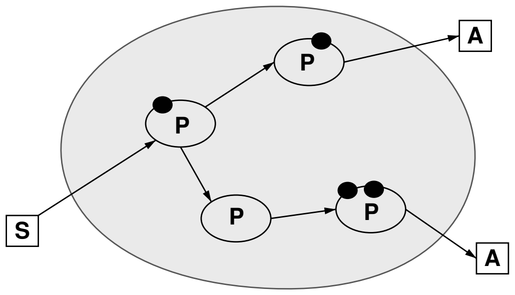

Figure 1. Solar consists a set of functionally equivalent nodes, named Planets (denoted as P), which peer together to form a service overlay. They connect data sources S and applications A and cooperatively execute context-fusion operators (filled circles).

All Solar sources register a name with a Solar's distributed directory service; Solar applications select data sources by querying the directory [10]. Since an operator is also a data source, it may optionally register a name, so it can be selected and shared by multiple applications. This modular approach encourages both code re-use (through documented class library) and instance re-use (through naming and discovery). Solar thus has the advantages of facilitating development of context-aware applications with a composable framework and increasing system scalability by reducing redundant computation and network traffic.

2.2 Planetary overlay

Solar consists of a set of functionally equivalent hosts, named Planets, which peer together to form a service overlay using a peer-to-peer protocol [17] as shown in Figure 1. A data source may connect to any Planet to advertise its availability and an application may connect to any Planet to select data sources and aggregate context with specified operators. The Planets cooperatively provide several common services including operator hosting and execution, a directory for source registration and discovery, an application-level multicast facility, policy-driven buffer management for flow/congestion control, a Remote Procedure Call (RPC) service, a distributed persistence service, and a dependency management service.

The directory service takes name advertisements, or name queries, encoded as a list of attributes [10]. The directory also allows an application to register a standing query and receive notifications about name space changes. Solar's multicast service follows a channel-based publish/subscribe model. A publisher simply sends events into a multicast channel, identified by a key, while subscribers register with the channel by supplying their own keys. Our implementation is based on Scribe [7], with an additional facility that uses application-specific policies to prevent buffer overflow [9].

2.3 Peer-to-peer routing

Solar employs a peer-to-peer (P2P) routing substrate, named Pastry, whose details are described in [17]. Here, we provide a brief overview to understand how we implement dependency graph management.

In Pastry, numeric keys represent application objects and are chosen from a large identifier space. Each Planet in Solar is a participating node (peer), which is assigned a key chosen randomly with uniform probability from the same key space. Pastry assigns each object key to the live node whose key is numerically closest to the object key. It provides a primitive to send a message to the node that is responsible for a given key. Each Pastry node also maintains keys for a set of their neighbors (in network proximity), called its *neighbor set*.

The overlay network is self-organizing and self-repairing, and each node maintains a small routing table with $O(log(n))$ entries, where n is the number of nodes in the overlay. Messages can be routed to the node responsible for a given key in $O(log(n))$ hops. Simulations on realistic network topologies show that: 1) the delay stretch, i.e., the total delay experienced by a Pastry message relative to the delay between source and destination in the underlying network, is usually below two; and 2) the paths for messages sent to the same key from nearby nodes in the underlying network converge quickly after a small number of hops [6].

3 Service design

In this section, we present the system design details of Solar's dependency service, which is used to manage components of the context-fusion framework, including both external clients (data sources and applications) and operators hosted by Planets. Table 1 lists explanations for the notations used in this section.

In following discussion, we indicate implicitly that a message is delivered through the Pastry P2P routing mechanism if the destination is a key. Otherwise, the message is delivered through a direct UDP/IP connection if the destination is an IP address. There is no delivery guarantee in either case.

X, K_X	A component and its key
R_X, M_X	A component's root and monitor
P_X, P_{R_X}, P_{M_X}	Planet hosting X, R_X, and M_X
P, P_M	Planet and its monitoring peer
C, CP	A client component and its proxy

Table 1. List of notations used in discussion.

3.1 Component registration

Before using the dependency service, a Solar component (whether a source, an operator, or an application) must first register with the service. When registering, a component X provides its key and configuration information as follows: 1) the action to take if X fails, such as to restart it or to email an administrator; 2) the command (or object class and initialization parameters) used to start X; 3) any restriction regarding the set of hosts where X be restarted; and 4) whether Solar can reclaim X when it has no dependents for a certain period of time. Solar records such configuration information and makes it available when X failed.

Some component may not be restarted on just any host. For instance, a data source may have to run on a particular type of host, to access a piece of sensing hardware. On the other hand, most operators are self-sufficient, processing events as they arrive, so Solar may restart them on any available Planets when they are lost.

Some components maintain state during operation, and require that state to be restored after a crash. We assume the component may checkpoint its state at a different host (using Solar's distributed persistence service, for instance) so the execution state is available during recovery. This state management is beyond the scope of this paper.

Solar may also migrate a running operator to another Planet. Solar implements a weak mobility scheme for operator migration, namely, it asks operator to capture its state that will be restored at the destination host [4].

Solar requires each component to explicitly identify the set of components it depends on; since dependencies may vary with the circumstances, components register (or remove) dependencies whenever necessary. A component may specify two types of dependencies: *key-based* or *name-based*. In other words, a component may specify the keys of the components it depends on, or a name query that will be resolved to discover components, who supply their keys in name advertisements. Since our directory service may return multiple components for a query, the requesting component may use a customized function to select appropriate components. For instance, a location-aware application may want to use the location service with maximum granularity or fastest update rate. As data sources come and go, as is typical in a ubicomp environment, the results of the query change occasionally; the function is re-evaluated, permitting quick adaptation for the dependent.

For either type of dependency, the dependent supplies a policy determining how to handle the failure, restart, or migration of the other component, or (for name-based dependencies) a change in the results of the name query and selector function. For example, if X depends on Y and Y fails, the policy of X may be to wait until Y is restarted. If the $X \to Y$ dependency is name-based, another reasonable policy is to use the output of the function to select a different component.

When a component has zero dependents, the component may be subject to garbage collection to release occupied resources.

In summary, a component registers with Solar's dependency service to provide its key, information about its restart configuration, and a list of its dependencies and associated policies. Solar monitors the component and restarts it accordingly; it also tracks the state of its depending components and takes appropriate actions stated in its policies.

3.2 Monitoring and recovery

If an operator X failed due to its internal exception, its hosting Planet can simply restart it. To recover X from a Planet failure, we install a dedicated monitor M_X as X's watchdog. The X periodically sends a *token* message to M_X through direct IP, and M_X assumes X is lost if it has not heard any token for a threshold of time. Then M_X selects a Planet to restart X, and the new X will register with dependency service and send tokens to M_X. On the other hand, M_X also periodically sends a token to X for monitoring purpose. So X could also detect the failure of M_X and restart it at another Planet. This bi-directional monitoring ensures both X and M_X can be restarted on new Planets unless they failed simultaneously.

We could have installed one monitor per component, but that would incur a large amount of monitoring traffic given many operators and relatively few Planets. It is a waste of both CPU power and network bandwidth. Thus we group the operators on the Planet and monitor them as a whole, so we only need one monitor for the Planet and the monitor restarts all the operators in the group when that Planet failed. As shown in Figure 2, the black circle is a representative of Planet P and sends aggregated tokens to monitor P_M for all the operators on P. Unlike operators that are hosted by Planets, an external client C (data source or application) installs a proxy operator CP on the Planet P that currently services C.

At the starting up, a Planet P first tries to find another Planet as its monitor, by sending a request to a random key

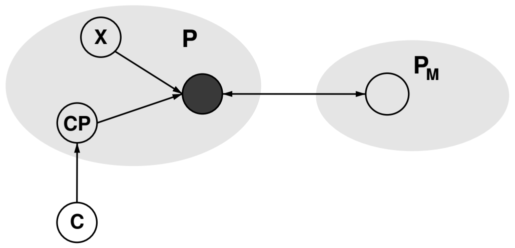

Figure 2. Component monitoring diagram. The black circle represents Planet P and sends aggregated tokens to monitoring Planet P_M. Unlike operator X, C is an external client that installs a proxy CP on the serving Planet.

through P2P and the receiving Planet responds with its IP address. When the request successfully returns, P starts to send an aggregated token, including the key and configuration of its operators, to its monitoring Planet P_M using direct IP connection. On the other hand, P_M will also send an acknowledge token to P at a fixed rate. Both P and P_M maintain a timer that is updated whenever a token is received from its peer and is expired when it is not updated for certain time.

Let the token sending interval at P be t, we assume if P_M has not received any token for a period of $k * t$, then P has crashed. Similarly, P assumes P_M has lost if it has not received any token for that threshold. By tuning the parameter k, the protocol becomes more resilient to intermittent token lost or more quickly to recover from failures.

As the timer at P_M expires, it starts the recovering process of all operators hosted originally by P. It sends a request to a random key whose responsible Planet, with the request containing the key and configurations of those operators. The receiving Planet restarts all the operators and make them join the local ones. Then P_M removes P from the list it monitors. If P detects that P_M has failed, it sends a monitoring request to a random key until a Planet other than itself is found.

To move an operator X from Planet P to P', P first removes X from its local repository so the token sent to P_M no longer contains X. The P also sends an explicit request to P_M to remove X's key and configuration. Then P requests P' to install a new copy of X. If either P or P' failed during migration process, P_M eventually times out and triggers process of recovering X.

We now discuss how to monitor and recover an external client C. The idea is similar, client C and its serving Planet P run a dual monitoring protocol that detects each other's liveness. If C has failed, P tries to restart it if C's configuration contains instructions. Otherwise, it simply removes

CP and de-registers C.

On the other hand, the proxy operator CP registers with dependency service and is also being monitored by P_M. When P has failed, P_M does not try to recover CP but simply removes CP from the list of operators to be restored. If C is still alive while P has failed, C may go through a discovery protocol to find another Planet for service.

By aggregating operators on the Planets we can significantly reduce monitoring traffic and processing overhead. A disadvantage of this approach, however, is that we are dumping all operators from the failed Planet to another, which may already be experiencing heavy load. In this case, we could either rely on Solar's load balancing mechanism to migrate some operators out later, or the monitor of the failed Planet may decide to partition the operator list and spread them onto several Planets.

Note the P and P_M form a two-node ring to monitor each other. If both P and PM failed simultaneously, our protocol is not able to recover the lost operators on P. Assuming a Planet fails with probability p and there is no co-related failures, the probability of simultaneous failure is p^2. We can further reduce this probability by inserting more monitors to expand the circle. Each monitor P_M periodically sends a token to its adjacent neighbor on the ring, which effectively monitors P_M and restarts it when it failed in a similar way to previous discussion. The token contains current addresses of all participating monitors, so a failure of non-adjacent members does not break the ring. With m nodes in the circle, the protocol failure probability exponentially reduces to p^m, with cost of linearly increased total token rate to be $m * r$ (assuming r is the token rate on one edge).

3.3 Tracking dependencies

To facilitate coordination between component X and those it depends on, we define a *root* object for X and denote it R_X. The root always runs on the Planet responsible for X's key, so any party can communicate with X by sending messages to K_X without knowing its network address. R_X tracks the current location of X, and forwards the message to X. X's monitor retains X's restart configuration, and the root R_X keeps X's dependency policies by receiving periodic updates from X as shown in Figure 3. Here we assume X depends on Y.

Upon receiving one of X's periodic update messages, R_X records X's current location and list of dependencies. If any dependency is name-based, R_X queries the directory service and evaluates the selector function on the results. The output determines the component(s) X should use; X is notified about any changes in the selection and X updates R_X if it indeed made those changes. Thus R_X contains a list of keys of the components X currently depends on, either explicitly specified or resolved from the name query.

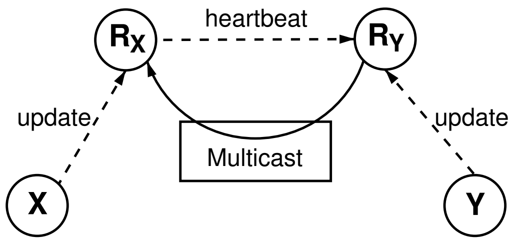

Figure 3. A component X has a root R_X that manages its dependencies. The root tracks the state of its component and publishes the events to its dependents' root, which in turn takes actions specified in dependency policies. The communication is based on P2P keys.

Root R_X receives notification whenever X fails, restarts, or migrates from either P_X or M_X. Then R_X publishes these events through multicast service to channel $g(K_X)$, where g is a deterministic mapping function to return the key for that channel. The R_X itself, subscribes to all the channels of the components X depends on, such as $g(K_Y)$. These events trigger R_X to take actions specified in X's dependency policies, for instance, to re-evaluate a selector function to get other usable components if Y has failed, or request P_X to reboot X if Y has rebooted. Note for scalability reasons, R_X does not keep the keys of its dependents. Instead, it simply publishes events to a single channel subscribed by X's dependents.

Given the list of keys X currently depends on, R_X sends a periodic *heartbeat* message to all the keys through P2P at a low frequency. The root of each components receives the heartbeat and resets its timer; the timer fires when it has not heard any heartbeat from any dependents for a long time. Then that root, say R_Y for component Y, assumes there is no dependents for Y anymore. If Y's configuration permits, R_Y requests P_Y to stop and de-register Y from dependency service.

Note the state kept by each root is soft and can be recreated from its component's periodic updates. This soft state helps the situation when P_{R_X} crashes, or R_X has moved to a different Planet due to overlay evolution. If the R_X times out on X's updates, it simply removes itself.

4 Implementation

Solar is implemented in Java 1.4.1 Standard Edition [1] and we adopt FreePastry [2] as its peer-to-peer routing substrate. In this section, we present the architecture of an

[1]http://java.sun.com/j2se/

[2]http://freepastry.rice.edu/

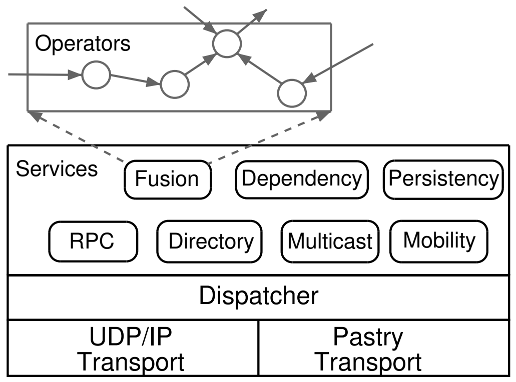

Figure 4. Architecture diagram of a Planet, which exposes two kinds of transport interface: normal socket communication and peer-to-peer routing. The dispatcher multiplexes received messages to a set of Solar services.

Planet and some optimizations to the dependency monitoring and recovering protocol.

4.1 Planet architecture

Planets are functionally equivalent and they all participate the overlay network, each acting as a Pastry node and having a unique key. A Planet has two kinds of transports: normal UDP/IP interface and Pastry interface. Thus a component running on Planet may send message with destination specified either as socket address or a Pastry key. A dispatcher routes messages from the two transports to a collection of Solar services based on the multiplexer header. From a service's point of view, it always sends messages to its peer service on another Planet. A service may also get a handle for another service on the same Planet and directly invokes its interface methods.

At starting up, a Planet loads the set of services specified in a configuration file. In particular, the "fusion" service manages local operators and schedules their execution. It uses directory service to connect operators to desired sources or other named operators. Solar disseminates events through operator graph using an application-level multicast facility to improve overall scalability. The "dependency" service implements the functionalities discussed in this paper and is used by fusion service to manage operator graphs.

4.2 Protocol optimizations

Here we discuss two optimization techniques on the protocols presented in Section 3.

In many cases we desire X be restarted on a Planet near original P_X to keep network proximity to other components. To achieve this goal, we add one more field in monitoring tokens containing a set of neighbor nodes NB_X of P_X. When P_X fails, M_X may request a random Planet from NB_X to start X instead of sending request to a random key. It is possible to make M_X run on a nearby Planet to P_X in same approach, but with increased possibility of that P_X and P_{M_X} fail together if we consider co-related failures in reality.

We can user another optimization to further reduce the traffic of monitoring and recovery by aggregating protocol messages that are being sent to the same IP or Pastry key. There are two places where the dependency service can group the messages. A Planet as the message originator may group token messages and heartbeat messages sent to the same IP address or Pastry key. On the other hand, a Planet as an intermediate node on the transmission path may check the destination key of passing by messages and group them by keys. Simulations on realistic network topologies show that the paths for messages sent to the same key from nearby nodes in the underlying network converge quickly after a small number of hops [6], thus the message aggregation may significantly suppresses protocol overhead.

5 Experimental results

In this section we present some results on monitoring and migration protocols. We performed the experiments on seven Linux-based workstations, [3] all being connected using a 100Mb switch. The average round-trip delay between any two hosts is about 0.25ms. On each workstation, we run a single copy of Planet that hosts a number of operators. We set the token rate to be once per 3 seconds, and the monitoring timeout to be 9 seconds. The root update rate is also once per 3 seconds.

First we measure how long it takes Solar to recover from a Planet crash and restore the operators on another Planet. We deliberately crashed a Planet and compute the time that its monitor detected the failure until the monitor's recover request returned. The recover request was sent to a random key whose responsible Planet will then re-initiate the operators from the crashed Planet and registers them with local dependency service. We show the result in Figure 5, with recovery time measured against various number of operators on the crashed Planet. We notice the time grows linearly as the number of operators to recover increases.

We then measure the time it takes for an operator to migrate from one Planet to another. In this test, every 5 seconds each Planet moved a random operator it currently hosts to another Planet. The migration first requested current Planet to remove the information about the moving op-

[3]Dell GX260, 2.0 GHz CPU, 512 MB RAM, and running RH Linux 9

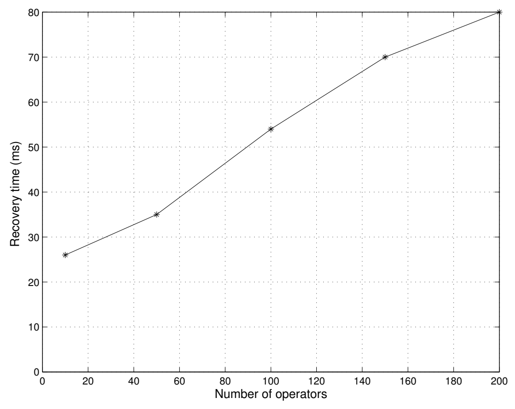

Figure 5. Operator recovery time in milliseconds after a Planet crash.

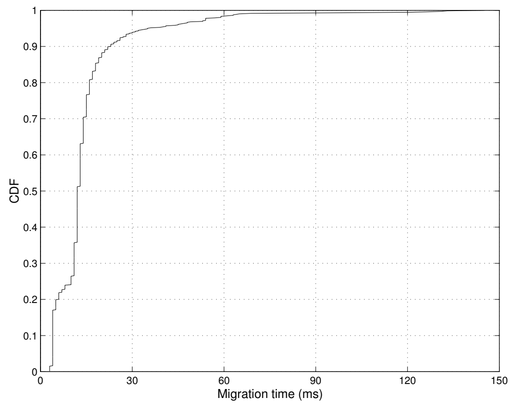

Figure 6. Distribution of operator migration time for 10-minute run.

erator, and then send the migration request to a random key. The Planet received the request restarts the operator using the configuration in the request, and the operator joined local ones to be monitored together. We measure the time between a migration request was issued until it successfully returned to original Planet. We recorded the delay numbers on all seven Planets during about 10-minute run, and Figure 6 shows the distribution. The median migration time is about 12 milliseconds. The more hops (P2P) migration request had to go through, the more latency it incurred. We also saw a long tails indicating a couple of 120 milliseconds delay, which might be caused by combined effect of large number of hops and thread scheduling of destination Planet.

6 Related work

The concept of data fusion is essential for component-based context-aware systems, such as Context Toolkit [12] and ContextSphere [11]. Until now, we have not seen a general service, like Solar provides, to manage the dependencies between the distributed data-fusion components. We believe our service abstracts away many complexities for coordination in a heterogeneous and volatile ubicomp environment.

Our dependency service mainly concerns the temporally-coupled components, and does not directly apply to other coordination models [3]. For instance, Stanford's Intelligent Room system provides temporally-decoupled communication over a tuple space [14]. Here we can only say a component depends on the tuple-space service while each component may be individually monitored and recovered. The functional and restart dependencies between components, however, are not clearly defined and may not be suitable for Solar to manage.

Recovery oriented computing (ROC) takes the perspective to reduce recovery time and thus offer higher availability [15]. One ROC technique is recursive restartability (RR), which groups components by their restart dependencies instead of functional dependencies [5]. When error or malfunction is detected, including component failure, RR system proactively restart the minimum component group containing the offending one. Our approach has a smaller scope and focused on the distributed dependency management protocols in case of failures. Solar supports restart dependency, in addition to functional dependency, by requiring explicit component registration.

Solar uses a rendezvous point in the infrastructure to manage a component's dependencies. Given a relative stable overlay network and the soft-state based root, the root becomes a natural entry point for inter-component coordination. This indirection-based technique has been used to manage large-scale event multicast [7] and host mobility [20].

INS provides a different communication primitive than P2P routing systems [1]. The destination of the messages is a name query, which is resolved hop-by-hop by the directories on the passing nodes. Thus INS is more expressive on message receivers, but may be less efficient compared to numeric key based routing. In particular, each service registers a name together with a metric value. As INS node resolves name queries, it may pick the name with smallest value. This is analogous to our approach to continuously evaluate a customized function over both the name advertisements to decide which components for use.

7 Conclusion

The contribution of this paper is a general service that manages distributed component dependencies. The goal of such a service is to abstract away most complexities encountered in a heterogeneous and volatile ubicomp environments, thus to ease the application or system design and implementation. We present a flexible ubicomp system platform, named Solar, which contains a service overlay built upon self-organized and self-repaired peer-to-peer routing protocol. Dependency management is one the services provided by Solar, and also uses Solar's directory and multicast services. In particular, we present how a context-fusion service on Solar uses dependency service to manage the operator graphs.

The dependency service contains two parts. One is to proactively monitor Planets among themselves and recover the lost operators on failed Planet. For each component, we maintain a rendezvous point (root) to manage its inter-component dependencies. A component registers with dependency service about its restart configuration, which is used by monitor to restart failed component, and dependency policies, which is used by root to take appropriate actions when depending components failed, rebooted, and moved. Our experiments show that the dependency service incurs small overhead on both recovery and migration protocols.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 186–201, Charleston, South Carolina, United States, 1999. ACM Press.

[2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[3] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *IEEE Computer*, 33(2):82–89, February 2000.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Weak and strong mobility in mobile agent applications. In *Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java*, Manchester, UK, Apr. 2000.

[5] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 125–130, Elmau, Germany, May 2001.

[6] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Topology-aware routing in structured peer-to-peer overlay networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8), Oct. 2002.

[8] G. Chen and D. Kotz. Context Aggregation and Dissemination in Ubiquitous Computing Systems. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 105–114, Callicoon, New York, June 2002. IEEE Computer Society Press.

[9] G. Chen and D. Kotz. Application-controlled loss-tolerant data dissemination. Submitted to Mobisys 2004, November 2003.

[10] G. Chen and D. Kotz. Context-Sensitive Resource Discovery. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 243–252, Fort Worth,Texas, March 2003.

[11] N. H. Cohen, H. Lei, P. Castro, J. S. Davis II, and A. Purakayastha. Composing Pervasive Data Using iQL. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 94–104, Callicoon, New York, June 2002. IEEE Computer Society Press.

[12] A. Dey, D. Salber, and G. D. Abowd. A context-based infrastructure for smart environments. In *Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments*, pages 114–128, Dublin, Ireland, Dec. 1999.

[13] J. Hightower, B. Brumitt, and G. Borriello. The Location Stack: A Layered Model for Location in Ubiquitous Computing. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 22–28, Callicoon, New York, June 2002. IEEE Computer Society Press.

[14] B. Johanson and A. Fox. The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 83–93, Callicoon, New York, June 2002. IEEE Computer Society Press.

[15] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report CSD-02-1175, U.C. Berkeley, 2002.

[16] B. Raman and R. H. Katz. Load balancing and stability issues in algorithms for service composition. In *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1477–1487, San Francisco, CA, April 2003. IEEE Computer Society Press.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 2001 International Middleware Conference*, pages 329–350, November 2001.

[18] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[19] A. Vahdat and A. Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proceedings of the 2nd USENIX Symposium of Internet Technologies and Systems*, Boulder, CO, October 1999.

[20] S. Zhuang, K. Lai, I. Stoica, R. Katz, and S. Shenker. Host Mobility Using an Internet Indirection Infrastructure. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, pages 129–144, San Francisco, CA, May 2003. USENIX Association.