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

1 Introduction

Applications in a computation and communication saturated pervasive-computing environment have to adapt their behaviors according the context of the user and the physical space. These "context-aware" applications typically obtain high-level situational understanding by aggregating one or more outputs of distributed sensors.

To reduce the complexity of developing and deploying these adaptive applications, we designed and implemented a flexible software infrastructure, *Solar*, for large-scale context fusion [3]. Solar consists of a set of functionally equivalent nodes, named *Planets*, which peer together to form a service overlay. A sensor connects to a Planet to register a name with Solar's *naming service* and publish a stream of *events*, while an application connects to a Planet to subscribe to one or more sensor outputs by querying Solar's name space. In addition, applications may inject data-fusion *operators* into the Solar to reduce the computation and communication costs on the mobile devices.

In this paper we present our approach to address the dependency management issues in Solar. Solar's *components* are sensors (their software processes), operators, and applications. We say a component X *depends* on another component Y if X needs events output from Y, and we say X is a *dependent* of Y. There are three problems: 1) to proactively monitor and recover lost components for their dependents; 2) to continuously evaluate application Z's queries over the dynamic name space to select best sensors Z depends on; and 3) to garbage collect operators whose dependents has departed, to reclaim occupied resources.

Solar employs a peer-to-peer (P2P) routing substrate, named Pastry [4], in which each participating node (Planet) is assigned a key chosen randomly with uniform probability from a large identifier space. Each component in Solar has a unique key from the same space. In addition to normal IP transport with socket address as destination, Solar also provides a primitive to send a message to a key K based on Pastry's router. The message eventually reaches the Planet whose key is numerically closest to K among all live Planets. We say this Planet is *responsible* for the key K.

2 Component registration

In Solar, a component must register by providing its key and a *configuration* 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 certain time.

Some components may not be restarted on just any host. For instance, a sensor may have to run on a particular 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 Planet when they are lost. Solar may also choose to migrate a running operator to balance load.

Solar requires each component to explicitly identify the set of components it depends on. 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. Since our name service may return multiple components matching a query, the requesting component may use a selection function to select appropriate components. When data sources come and go, as is typical in a pervasive-computing environment, the results of the query change occasionally and the selection function is re-evaluated, permitting quick adaptation for the dependent [1].

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 selection 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 $X \rightarrow Y$ dependency is name-based, another reasonable policy is to use the output of the selection function to choose a different component.

3 Monitoring and recovery

When a Planet P starts life, it finds another Planet P_M as its monitor by sending a request to a random key. The

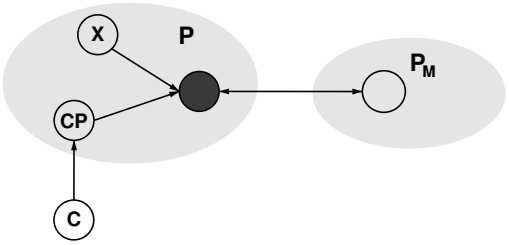

Figure 1. Component monitoring diagram.

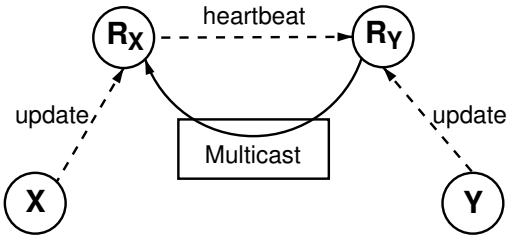

Figure 2. Dependency tracking diagram.

P registers the configurations of all its operators with P_M, and these two Planets engage in a heartbeat protocol between themselves. If P has not heard an update from P_M for a certain time, it assumes P_M has failed and finds another Planet as its monitor. If P_M has not heard an update from P for a certain time, it assumes P has failed and tries to restart all the operators originally running on P.

We now discuss how to monitor and recover an external client C (either a sensor process or an application). The idea is similar, client C and its serving Planet P run a dual monitoring protocol that detects each other's liveness. As shown in Figure 1 the client C installs a proxy CP on its serving Planet. 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 CP 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.

4 Tracking dependencies

In Solar, a component X may move from one Planet to another at runtime. We define a *root* object for X and denote it R_X, which tracks X's current location. The root always runs on the Planet responsible for X's key, so X's dependents can find X's location by simply sending a query to K_X. Root R_X receives periodic updates from X about its current location and dependency policies, as shown in Figure 2. 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 naming service and evaluates the selection 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, whether explicitly specified or resolved from the name query.

Root R_Y receives notification whenever Y fails, restarts, or migrates from either Y's hosting Planet or its monitoring peer (P and P_M). Then R_Y publishes these events through Solar's multicast service to channel T_Y [3]. Since R_X subscribes to the channels of the components X depends on, such as T_Y, these events trigger R_X to take actions specified in X's dependency policies, for instance, to re-evaluate a selection function to get other usable components if Y has failed. Note that, for scalability reasons, R_Y does not keep the keys of Y's dependents. Instead, it simply publishes events to a single channel subscribed by Y's dependents.

Given the list of keys X currently depends on, R_X sends a periodic *heartbeat* message to all the keys through Pastry at a low frequency. The root of each component 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 are no remaining dependents for Y. If Y's configuration permits, R_Y asks P_Y (the Planet currently hosting Y) to stop and de-register Y.

5 Summary

Our approach to dependency management in Solar requires each component to explicitly register a *recovery configuration* and all *dependencies*. Each Planet P has a peer monitoring Planet P_M that is responsible to initiate recovery, according to component configurations, if P_M detects the failure of P. On the other hand, the Planet responsible for the key of a component X tracks all the components X depends on and implements the associated policies when those components fail or when the result of X's name query and selection function change. We present the details of our protocols and experimental results in an extended report [2].

References

[1] 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.

[2] G. Chen and D. Kotz. Dependency management in distributed settings. Technical Report TR2004-495, Dept. of Computer Science, Dartmouth College, March 2004.

[3] G. Chen, M. Li, and D. Kotz. Design and implementation of a large-scale context fusion network. Submitted to MobiQuitous 2004, February 2004.

[4] 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.