1.1 Blogjecting Watchdog

Achieving availability is a multi-layered effort. I've already talked about how services should be autonomous (see for example Active Service pattern in chapter 2) , the Blogjecting Watchdog pattern will take a look at another aspect of autonomy. The Blogjecting Watchdog pattern shows how a service can proactively try to identify faults and problems and to try to heal itself when it identifies these problems.

1.1.1 The Problem

The Service Instance pattern (see section 3.4) for example, demonstrates a strategy that a service can implement to be able to cope with failure. The question is – is that enough? Is it enough for the service to try to cope with everything by itself? My answer is no, that is not enough. For one once we dealt with the failure within the service, the service ability to cope with the next failure would probably be diminished. For example if we found a failure in a server and moved to a standby server, the new server does not have another stand-by server to move to if another fault occurs.

Additionally, the failure might be too much for the service to be able to overcome it by itself. Like a switch going down - So we would have something external that looks after the service and could help the service (see Service Monitor pattern in chapter 4).

To increase the service autonomy and to increase the overall availability of our SOA we need both to try to identify and repair problem and to be able to notify the world about the service's current status.

The question is then:

> **How can we identify and attend to problems and failures in the service and increase service availability?**

One option is to try to infer the state of the service from the way it looks to the outside – yes this is as crude as it sound. You try to call the service, it doesn't respond you know it is down; you call the service, you expect to get a reply in 5 seconds you get it in 10 seconds, you understand that the service is congested. This is not a very good option as the external behavior only gives us coarse knowledge on the service's state. For example, if the services has a decent fault tolerance solution, we wouldn't know that anything happened – but the truth is that the service ability to handle the next fault might not exist anymore.

Another way is to install agents on the service's servers, this will give you a much better picture of what happens (vs. the option above). For example, you will also be able to get trend information (e.g. You can watch how much disk space is left and alert when it is getting low). There are several problems with

this solution. One is that you need to actively install software on the service's servers which both decreases the service autonomy and creates a management hassle in itself. Another problem is that you still only get an external view of the service behavior (you just gain access more information). There are situations (see for example the Mashup pattern in chapter 7) where not all the services are under your control and you cannot access their hardware.

Yet another option is to actively question the service about it state. The has one big advantage over the two previous options since you also get some inside information regarding what the service has to say about its state. This enables the service to communicate trends in problems that will actually make it fail. For example if the service does not write any information into the local disk a low disk space is not a problem at all, if this is the disk where the database is located it is very much a problem. The solution is not perfect since it is the observers responsibility to go after the information. If the rate at which the observer samples the service is not fast enough it can miss on vital information.

As I mentioned earlier we want something that will help increase the service's autonomy so a better approach in this regard would be for the service to watch over itself

1.1.2 The Solution

Watching over itself is also not enough as we also said we need the "world" to know what happening with the service, thus a combines solution is to :

> **Implement the Blogjecting Watchdog pattern and have the service actively monitor its internal state, try to heal itself and continuously publish its state and other important indicators.**

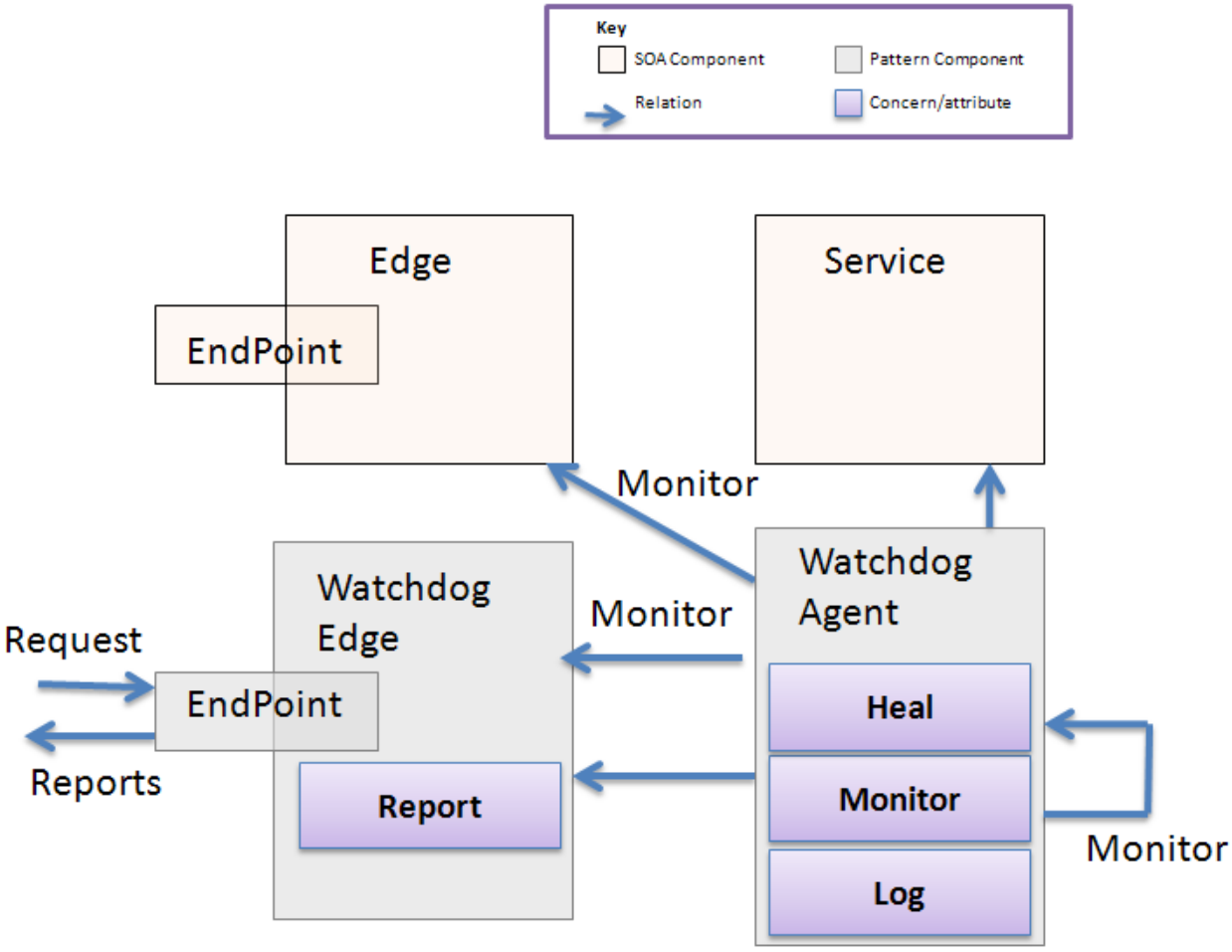

Figure 3.14 The blogjecting watchdog pattern. The blogjecting. The blogjecting component that send the reports out and and listens for requests. The watchdog component monitor the status of the business service, tries to heal stray components and log any failure.

The pattern revolves around a single idea – to increase the service responsibility by using two complementary concepts reporting and self healing. The first is the Blogjecting concept where the service implements the Active Service pattern (see chapter 2 for more details) and a component which is in charge of monitoring the service's state. The component publish (see the publish/Subscribe interaction pattern in chapter 6) also the service's state on a cyclic basis or when something meaningful occurs. It is important to note that the fact that the service actively publishes its state doesn't have to mean it cannot also respond to inquiries regarding its health (akin to living a comment on a blog and getting a response from the author)

What are Blogjects

The term Blogjects was coined by Julian Bleecker back in 2005 (Bleecker, 2005) to describe "edgy designed objects that report themselves, or expose their experiences in some fashion" or in other words Blogject == Objects that blog. Julian Bleecker's vision for Blogjects is wider than the one suggested here. Jonathan's vision is for things that participate in the Web 2.0 sense of social-web or even further than that – to use Julian's words :"Forget about the Internet of Things as Web 2.0, refrigerators connected to grocery stores, and networked Barcaloungers. I want to know how

to make the Internet of Things into a platform for World 2.0. How can the Internet of Things become a framework for creating more habitable worlds, rather than a technical framework for a television talking to an reading lamp?" . I highly recommend taking a look at the full paper "A Manifesto for Networked Objects – Cohabiting with Pigeons, Arphids and Aibos in the Internet of Things" (Bleecker, 2006) to get the full picture.

The second concept that plays in the Blogjecting Watchdog pattern is the watchdog, The idea here is to have a component that listens in on the information gathered and published by the blogject component and then to acts on that information in a meaningful way to increase the reliability and availability of the service. The possibilities for implementing self-healing are endless, two simple examples for self-healing actions are restating failed components and cleaning temporary files.

Watchdogs

Watchdog (actually watchdog timer) is a term borrowed from the embedded systems world. A watchdog is a hardware device that counts down to zero, and when it gets there it reset the device. To prevent this reset the application has to "kick the dog" before the timer runs out. If the application does not reset the counter it means that the application is hanged and the idea is that the reset would fix that.

How is the Blogjecting Watchdog pattern better than the other options mentioned above?

Even if we just consider the blogjecting part of the pattern we can see several advantages over the other approaches. The Blogjecting Watchcdog combines the benefits of an agent that actively monitors the service's health with the internal knowledge of what's important for the service continuity and what's not. Unlike the external agents solution, using Blogjects, the service retains its autonomy. The autonomy is increased even further when you combine the self-healing features of the watchdog. Thus the end result is a service which is more resilient (and thus has higher availability), which lets the world know both its current state as well as future trends.

In one project I was working on we inherited a situation where there were interdependencies between executable installed on different servers (within a service) – for example when one process was down on server A the objects running on server B could not function well and other such dependencies (this isn't the brightest design, but sometimes you have to compromise - in this case there was no time and budget to redesign these applications). What we ended up with, is something like the situation in figure 3.15 below:

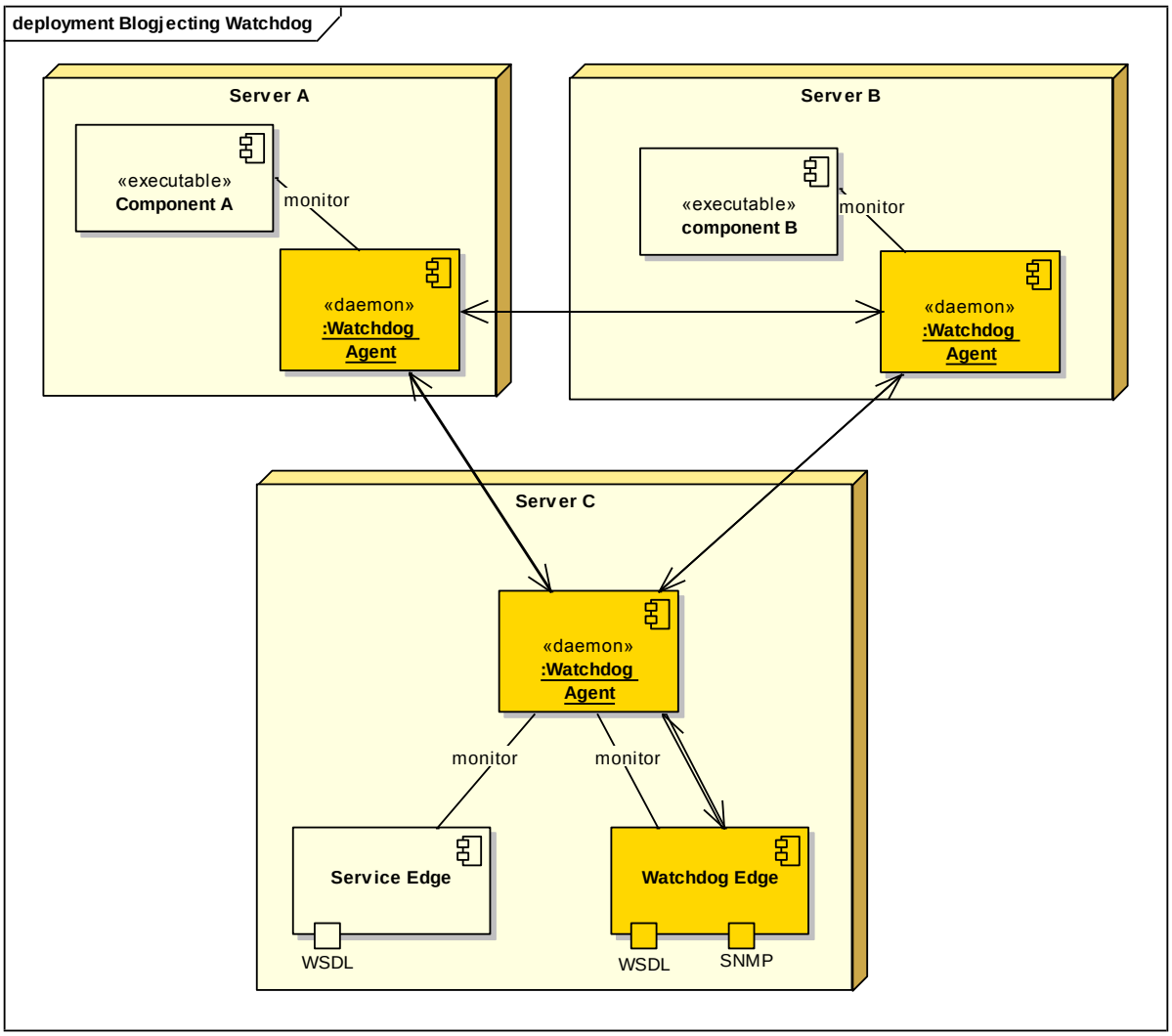

Figure 3.15 a sample deployment of a blogjecting watchdog. The daemons on the servers monitor the running components on each server. The Watchdog edge exposes the current the current state both through a web-services API and as SNMP traps

The watchdog agents on each of the server nodes monitors the components. The agents communicate amongst themselves to examine the dependencies and actions taken. The watchdog Edge component provides a WSDL based endpoint where other services can query it for the service's health. It also publishes SNMP traps to an external SNMP monitor (e.g. HP-Openview). As an implementation hint, I can suggest keeping the watchdog components in a separate very simple executable (preferably a daemon that runs when the OS loads). The simpler the component, the lower the risk it will fail in itself (you can of course have a backup in the form of a hardware watchdog ..). Let's take a more thorough look at the technology mapping options

1.2.1 Technology Mapping

Implementing Blogjecting Watchdog in an enterprise will usually pre-determine the protocols you will have to use for your "blog". The IT team will most likely already standardize on one of the leading monitoring suites (CA-Unicenter, HP-Openview, IBM-Tivoli or if you are an all Microsoft shop

Microsoft Operations Manager). In these cases you can use the SDK of the monitoring software (e.g. the Unicenter Agent SDK or MOM management pack developer guides). There are even 3rd party software packages to help you build such agents (for example OC Systems have a Universal Agent that makes it easier to write agents for Unicenter).

Note, that this is not always the case though, and sometimes you do have the freedom to choose you protocols. Few projects I worked on chose to standardize on using web-services with specific messages for monitoring the health of service (so we had a specific endpoint for each service where these messages were supported). With the emergent of SOA specific tools like the ones by Amberpoint and Weblayers you will see more and more WS-* based monitoring.

Other ways for reporting your internal state can be to use standards like SNMP (Simple Network Management Protocol) or plainly the windows Event logs An interesting option, which will let your Blogjecting Watchdog literally blog is to use a product called RSSBus. Whish is an ESB implementation that uses RSS protocol for communications. At the time I am writing this, the product is still in beta, so I haven't used it for a serious system yet. Nevertheless, it looks like an interesting direction which I'll consider when it is released.

Regarding the self-healing part (watchdog), self-healing is still more prevalent in hardware then in software (watchdog timers, RAID, IBM , hot spare memories, hot spare drives etc.) in a sense any solution that builds on clustering technology also has some of that built-in. The virtualization trend will also help in this sense (see discussion on utility computing in this chapter's summary). You can already read papers that talk about self-healing web services (G. Kouadri Mostéfaoui, 2006) or see some projects that tries to look into this problem (e.g. WS-Diamond - DIAgnosability, Monitoring and Diagnosis). Nevertheless, all of them are still in the research phase and if you want something now, you will probably need to implement something by yourself. In my experience, it won't take you too much time to have a basic watchdog up and running , but it will take you sometime until you will have it predicting and acting as an advanced warning system.

1.2.2 Quality Attribute Scenarios

The Blogjecting Watchdog is an interesting pattern (and not just because of its odd name) as it can really help on the way to autonomous computing. The effect of this proactive approach is to increase the overall reliability of the service. A service which is self-healing can overcome (at least) minor problem which results in better availability overall. Additionally the monitoring aspects of the Blogjecting Watchdog also help enhance availability by notifying administrators that something is amiss (which will enable them to fix it).

Quality Attribute (level1)	Quality Attribute (level2)	Sample Scenario
Availability	Failure detection	Upon a failure or degraded performance, The system will alert the system admin (via SMS) within 3 minutes.
Reliability	Increased autonomy	During normal operations, the system will clear all its temporary resources (e.g. files) continuously

Table 3.8 Blogjecting Watchdog pattern quality attributes scenarios. These are the architectural scenarios that can make us think about using the Blogjecting Watchdog pattern.

Once we introduce a monitor and start to collect data, we can start to find new uses for that data, for example we can use the information on incoming request to try to locate attacks on the service etc. Saved monitoring data can be used to analyze the service's behavior over time, predict failures and thus increase its maintainability etc.